Università degli Studi di Salerno

Dipartimento di Informatica

Dottorato di Ricerca in Informatica
Curriculum Internet of Things and Smart Technologies
XXXVI Ciclo

PH.D. THESIS

How Reusability Mechanisms and Built-in Functions Impact Code Quality Over Time

GIAMMARIA GIORDANO

SUPERVISOR:	PROF. FABIO PALOMBA
CO-ADVISOR:	PROF. FILOMENA FERRUCCI
PHD PROGRAM DIRECTOR:	PROF. ANDREA DE LUCIA

A.A 2022/2023

SOFTWARE ENGINEERING LAB, UNIVERSITY OF SALERNO
HTTPS://SESALABUNISA.GITHUB.IO/

This research was done under the supervision of Prof. Dr. Fabio Palomba and Prof. Dr. Filomena Ferrucci, with the financial support of the University of Salerno and Business Solution (BS) s.r.l. from February 1st, 2021, to January 31st, 2024.

The final version of the thesis has been revised based on the reviews made by Prof. Dr. Giuseppe Cascavilla and Prof. Dr. Fausto Fasano.

The doctoral committee was composed by:

Prof. Dr. Delfina Malandrino	University of Salerno, Italy
Prof. Dr. Sergio Di Martino	University of Naples "Federico II", Italy
Prof. Dr. Massimo Torquati	University of Pisa, Italy

First Release, March 2024. Revised on April 2024.

“Non conta quanto hai perso prima, nella tua vita [...]
C’è sempre un modo ed una chance ”.
Fabio Bartolo Rizzo - Marracash

RINGRAZIAMENTI

Alla fine del mio percorso di dottorato, ritengo opportuno guardare indietro e ringraziare coloro che mi hanno sostenuto durante questo periodo impegnativo.

La lista delle persone che dovrei ringraziare è molto lunga, ma cercherò di essere il più coinciso possibile.

Un primo ringraziamento va al coordinatore del dottorato, il **Professore Andrea De Lucia**. Mi dispiace per tutto il tempo che hai speso nel risolvere sia i miei problemi istituzionali che quelli burocratici. Se non sono finito in prigione in questi anni è solo grazie a te!

Un grazie di cuore anche ai miei advisor, il **Professore Fabio Palomba** e la **Professoressa Filomena Ferrucci**. Entrambi non si sono limitati a supervisionarmi con l'obiettivo di affinare le mie competenze accademiche, ma mi hanno anche fornito un supporto emotivo attraverso preziosi consigli. La loro esperienza, pazienza e incoraggiamento sono stati pilastri fondamentali di questo percorso, permettendomi di crescere sia professionalmente che personalmente.

Sono profondamente grato per la loro guida e supporto continuo.

Fabio, sono veramente grato che tu abbia condiviso le tue conoscenze con me. Attraverso la tua guida ho acquisito innumerevoli competenze, sia tecniche che non. Da te ho imparato come scrivere articoli scientifici, come effettuare revisioni e come selezionare le giuste conferenze o journal. Hai inoltre instillato in me la passione per l'accademia e mi hai aiutato a tracciare un potenziale percorso per il mio futuro, per questo ti sarò sempre riconoscente.

Ripercorrendo i vari tasselli che compongono questa tesi ho riletto i sorgenti LaTeX di tutti i paper che abbiamo scritto insieme. Ho rivisto tutti i feedback che mi hai fornito nel tempo, come "Questa sembra essere una descrizione di un exploit, non di una vulnerabilità" o "Questa sezione dovrebbe essere più focalizzata dal punto di vista dello sviluppatore". Ho osservato una notevole riduzione dei tuoi commenti negli anni, segno evidente di come la tua guida abbia affinato le mie capacità professionali. Il tuo approccio nei miei confronti ha seguito un chiaro percorso: inizialmente eri tu a scrivere intere sezioni dei nostri articoli, affidandomi il compito di rileggerli e fornire feedback. Gradualmente hai iniziato a delegarmi intere sezioni, fino a farmi redigere l'intera bozza del paper in modo indipendente. All'inizio, ammetto di essere stato intimidito dal compito, sentendomi non all'altezza; quindi mi sforzavo di emulare i tuoi costrutti sintattici e i tuoi stili. Ogni volta che facevi un "pass" sui nostri paper per "Palombizzarli", mi rileggevo il *prima* e il *dopo* per capire quali parti erano state modificate e come, così da apprendere il modo "giusto" di esprimere determinati concetti. Mi hai insegnato veramente tanto e spero di averti trasmesso e insegnato qualcosa anch'io.

Filomena, mi hai guidato non solo durante il mio percorso di dottorato, ma anche durante la mia tesi di laurea magistrale. Da te ho appreso competenze inestimabili, come gestire situazioni difficili e comunicare efficacemente con gli altri. Questi consigli sono stati essenziali per la mia carriera e per la mia crescita personale.

Ti sono profondamente grato per il tuo sostegno costante e per aver investito tanto tempo e impegno nel mio sviluppo. La tua guida è stata un faro di ispirazione; sotto la tua tutela ho acquisito non solo competenze professionali, ma anche un profondo senso di responsabilità ed etica professionale. La fiducia e la meticolosità che hai dimostrato nel tuo lavoro hanno stabilito uno standard a cui aspiro. La tua pazienza e comprensione, specialmente nei momenti in cui ho affrontato difficoltà, mi hanno fornito un ambiente sicuro e di supporto per crescere. Grazie ancora una volta per la tua dedizione.

Voglio anche ringraziare la **Professoressa Gemma Catolino** (a.k.a. Golli). Gemma, oltre ad essere una delle co-autrici di due dei paper di cui sono più orgoglioso, sei stata di grande supporto, specialmente durante l'ultimo periodo del mio percorso di dottorato. Mi hai insegnato che anche in accademia (come nella vita) le cose non vanno sempre come si spera, ma che è giusto battersi per cercare di cambiarle. Ti considero una guerriera capace di sollevare il mondo per combattere le ingiustizie. Credo sinceramente che tutto il *SeSa Lab* possa beneficiare non solo delle tue immense competenze e conoscenze, ma anche delle tue capacità di ascoltare e mitigare possibili conflitti. Sei inoltre una prof.ssa capace di trasmettere passione ai tuoi studenti (anche se non sempre ti piacciono gli argomenti che spieghi). Spero che nessuno ti faccia più arrabbiare, anche se lo vedo difficile (:-)). Prometto che un giorno faremo insieme un tool e lo chiameremo “Cringe (anche se non a tutti piacerà questo nome)”.

Come ogni viaggio che si rispetti, c’è sempre bisogno di qualcuno che ti riporti alla realtà. Nel mio caso (forse un po’ di tutto il SeSa Lab), questo ruolo l’ha ricoperto il **Professore Dario Di Nucci**. Dario, tu mi hai trasmesso il tuo spirito critico. Avere i complimenti da te è qualcosa di prezioso, poiché punti sempre all’eccellenza. Sono stato davvero fiero di me stesso quando dopo la presentazione ad *MSR*, mi dicesti “sei andato bene”. Per me quelle parole hanno rappresentato davvero un traguardo. Sono davvero orgoglioso di poter collaborare con te.

Desidero inoltre ringraziare il **Professore Fabiano Pecorelli** (a.k.a Pec). Fabiano, fin da subito ci siamo trovati in ottimi rapporti, forse anche per via della nostra vicinanza geografica. Da te ho imparato che se da un lato è giusto dedicarsi anima e corpo al lavoro, dall’altro è giusto svagarsi e prendere le cose più “alla leggera”. Credo che un professore come te possa ristabilire all’Università quel contatto con gli studenti che certe volte viene a mancare. Anche se in questi anni non abbiamo avuto modo di collaborare attivamente sono davvero felice di averti conosciuto e sono grato di tutti gli insegnamenti diretti e indiretti che mi hai trasmesso.

Tra gli ultimi dei “*big*” (ma non per importanza) c’è il **Professore Carmine Gravino**. Carmine, credo di non avertelo mai detto, ma è anche grazie al tuo supporto che decisi di intraprendere questo percorso. Durante il mio percorso di Laurea Triennale ti vedevo come il professore un po’ scontroso (forse non te lo ricordi, ma tu mi facesti anche l’esame orale di *Basi di Dati*). Durante la Magistrale mi facesti appassionare al corso di *Metriche e Qualità del Software* e decisi in quel momento che volevo a tutti i costi mettere in pratica quelle conoscenze. In quel momento non me ne accorsi, ma stavo forse decidendo la mia strada. Grazie di cuore sia per quello che mi hai insegnato, sia per le risate fatte insieme durante le conferenze (sopratutto negli eventi sociali).

Terminata la “lista dei grandi”, vorrei ringraziare tutto il *SeSa Lab* per ciò che hanno rappresentato per me in questo periodo.

Valeria, ancora una volta mi trovo a doverti ringraziare. Per un periodo pensavo che le nostre strade si sarebbero divise alla fine del nostro percorso Magistrale, ma in qualche modo siamo ancora qui. La distanza che c’è (geograficamente parlando) non ci ha divisi, anzi, ha in qualche modo rafforzato il nostro legame. Sono felice di averti avuto al mio fianco durante questo periodo, poiché ho sempre ricevuto da te supporto e conforto. Sono sicuro che farai molta strada in questo ambito, se lo vorrai, perché hai tutte le capacità necessarie.

Mi ricordo ancora quando insieme entrammo nell’ufficio di Filomena per chiedere informazioni sul percorso di dottorato e di tutte le preoccupazioni riguardo al futuro. Entrambi ci chiedevamo se fosse la scelta giusta applicare al bando poiché non ci ritenevamo all’altezza. Alla fine, non solo fummo selezionati, ma siamo anche arrivati alla fine (forse grazie alla tua capacità di cambiare la realtà a tuo piacimento).

Emanuele, inizialmente con te (come con Manuel) è stato difficile stringere un legame, poiché la pandemia ci “concedeva” solo qualche meeting a distanza, ma una volta incontrati di persona ho imparato ad apprezzarti. Sei una delle poche persone con cui condivido molte passioni “nerd” (ad esempio, Dragon Ball) e con cui posso fare battute un po’ “sopra le righe” senza risultare offensivo (:-)). Hai un talento innato per questo lavoro e sono sicuro che saprai cogliere l’occasione giusta quando ti si presenterà.

Manuel... beh, di te ci sarebbe tanto di cui parlare (commisurato almeno al numero medio di parole che sei capace di dire nel giro di un’ora). Anche con te inizialmente la pandemia ha tentato di tenerci distanti, ma alla fine, anche con te il rapporto che si è creato è indissolubile. Il SeSa Lab è sicuramente un posto meno rumoroso (anche se Dario Di Nucci prova a tenere alto il tuo onore) da quando ti sei dottorato. Ricordo ancora tutti i pranzi passati insieme, quando tentavi a tutti i costi di parlare di politica e noi puntualmente ti interrompevamo cercando di cambiare argomento. Spero che il percorso intrapeso possa portarti grandi soddisfazioni.

Ringrazio la mia fidanzata **Raffaella**, che in questi anni mi ha sempre appoggiato nelle mie scelte, motivandomi ad andare avanti e aiutandomi a prendere decisioni spesso difficili. Il tuo supporto incondizionato è stato-fondamentale nel mio viaggio personale e professionale. Mi hai (ancora una volta) sia supportato che sopportato, dimostrando una pazienza e una comprensione che hanno alleggerito i momenti di stress e incertezza. Sono sicuro che tutte le sfide che ho affrontato sarebbero state molto più complesse se non avessi avuto il tuo sostegno costante al mio fianco. La tua presenza ha trasformato i momenti difficili in opportunità di crescita e apprendimento condiviso e per questo ti sono profondamente grato.

Vorrei anche ringraziare tutti gli altri membri della grande famiglia del SeSa Lab: **Francesco** (Ciccio), **Stefano** (La scazzetta è solo una torta alle fragole), **Giulia**, **Vincenzo**, **Gabriele**, **Dario** (ma alla fine come ti sei rotto il dito?), **Gilberto** (Gibbo), **Giusy**, **Pir**, **Viviana**, **Gianmario**, **Alexandra**, **Antonio** (X2) e **Gerardo**. Molti di voi vi conosco da anni, altri invece da pochi mesi, ma sono sicuro che il nostro legame crescerà sempre di più!

Desidero inoltre ringraziare i miei amici da sempre, ovvero tutto il gruppo della "SIC" con cui ho passato gran parte della mia vita, ovvero **Salvatore**, **Cosimo**, **Giovanni**, **Raimondo**, **Vincenzo** e **Giuseppe**.

Ringrazio anche la mia famiglia, mia madre **Elisabetta**, mio padre **Rocco**, mio fratello **Benito**, la sua compagna **Anthea**. Siete stati sempre presenti e mi avete sempre incoraggiato. Grazie di tutto.

Ringrazio inoltre tutti i "componenti aggiuntivi" della mia famiglia, ovvero , il mio cane **Jack**, la mia gatta **Lucy**, la mia tartaruga e la mia cagnolina **Lilly** che mi guarda dall'alto.

Desidero ringraziare **Salvatore**, **Michela**, **Katia**, e **Alessia** che mi hanno accolto nella loro casa trattandomi come uno di famiglia fin dal primo giorno. Con voi ho passato momenti bellissimi. Grazie di cuore di tutto ciò che avete fatto in questi anni per me.

Un ultimo (ma non per importanza) ringraziamento va ai revisori di questa tesi di dottorato, ovvero il **Professore Fausto Fasano** e il **Professore**

Giuseppe Cascavilla. Le vostre revisioni mi hanno dato l'opportunità di migliorare l'intero elaborato, affinando la chiarezza espositiva e la precisione degli argomenti trattati. Oltre a ciò, i vostri consigli metodologici e le critiche costruttive hanno notevolmente arricchito la qualità della ricerca.

ABSTRACT

Nowadays, the adoption of software systems worldwide is there for all. Daily, end-users use software systems to perform a number of operations ranging from simple operations e. g., setting up a timer, to complex operations e. g., monitoring an environment using complex artificial intelligence (AI) systems. Despite the plethora of software systems worldwide, one of the key points that unite them is that they need to evolve to avoid premature obsolescence. However, frequent *change requests* during maintenance and evolution activities can compromise software quality due to sub-optimal design decisions or the pressure to meet tight deadlines, often leading to design erosion. As a result, much of the existing literature on code quality in software systems focuses primarily on aspects such as defect proneness, code smells, and maintenance effort. Nevertheless, key questions about how code quality evolves over time and the impact of software engineering practices—such as the adoption of reusability mechanisms or the utilization of built-in features of programming languages—during software maintenance and evolution remain unaddressed.

This thesis aims to bridge this gap by investigating how code quality evolves over time and examining the influence of software engineering practices on maintenance and evolutionary activities.

The *ultimate goal* of this thesis is to provide a step forward on the so-called *Evolutionary Code Quality* i. e., the quality attributes that developers need to monitor during software maintenance and evolution.

To address our objective, we performed mining software repository (MSR) studies on software systems, taking into account two aspects: I) the temporal variation of code quality and II) the relationship between reusability mechanisms and built-in features and code quality.

Our findings revealed that: I) reusability mechanisms exhibit temporal variability without following a discernible pattern; II) these mechanisms can often mitigate code quality concerns, particularly for code smells; III)

the peculiarity of built-in functions in software systems can stimulate the proliferation of specific code smells.

In addition, as the last part of this dissertation, we provide two further studies related to code quality by considering systems developed using emerging technologies (i. e., IoT systems and mobile apps) to give an overview of the main aspects that these systems need to consider related to code quality i. e., security and privacy aspects.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

PART I

INTRODUCTION AND BACKGROUND

1 INTRODUCTION

1.1 CONTEXT AND MOTIVATION

Nowadays, the perception of software systems' capabilities is rapidly changing. The introduction of new technologies to the market has shocked the world by drawing end-users attention to these capabilities. Applications such as ChatGPT[1], Google Gemini[2], or Meta LLAMA[3] are just an example of the abilities of software systems to perform special and general-purpose tasks.

Despite the differences among software systems (such as the environments where they operate or the features they offer), one common factor that characterizes them is the need to evolve and be maintained over time to avoid premature obsolescence [192]. We go back to the time between the 1960s and the 1970s, an IBM engineer i. e., Lehman, noted that the task of successfully evolving software systems was more complex than it appeared, and based on his experience, he formulated the eight empirical rules that are now known as "*Lehman's Laws of Software Evolution*" [121]. It is important to note that the laws proposed by Lehman were reworked until the 1990s.

L_1: Continuing Change. If the software is not continuously adapted to new needs, it will gradually become less satisfactory to its users, decreasing its effectiveness and use.

L_2: Increasing Complexity. A software system tends to increase in complexity over time unless deliberate efforts are made to reduce its complexity through specific activities.

[1] https://chat.openai.com/
[2] https://gemini.google.com/?hl=it
[3] https://llama.meta.com/

L_3: **Self-Regulation**. Software evolution functions as a self-regulating process, yielding a nearly normal distribution of both product and process artefacts.

L_4: **Conservation of Organizational Stability**. The average overall activity rate of an evolving software system remains consistent over time; i. e., the amount of work completed in each release remains relatively constant.

L_5: **Conservation of Familiarity**. The portion of new content in the next releases tends to be constant or to decrease over time.

L_6: **Continuing Growth**. The number of functionalities in software systems tends to increase over time.

L_7: **Declining Quality**. The user perception of system quality declines over time unless its design is not thoroughly maintained and adapted to the new operational constraints.

L_8: **Feedback System**. To effectively evolve a software system, it is crucial to acknowledge that the development process operates as a feedback system with multiple loops, agents, and levels.

Taking a closer look at Lehman's laws, especially focusing on the L_2 (Increasing Complexity) and L_7 (Declining Quality), it is possible to observe that both the rules implicitly suggest that the continuous *change requests* that software systems receive daily can gradually degrade software quality attributes, leading to more complex software maintenance and evolution. For this reason, due to the characteristics of software systems that are "by design" change prone [116], the literature predominantly focuses on quality aspects related to defect proneness [176, 253], code smells [85, 348], maintenance effort [318].

To facilitate the evolution and maintenance activities, the software engineering community proposes a number of mechanisms that practitioners can utilize. Among them, two well-established and adopted daily by practitioners are the reusability mechanisms and the use of built-in functions. The former are commonly used in object-oriented programming languages and refer principally to the use of design patterns [75], programming abstractions [166] (e. g., inheritance and delegation), or third-party libraries [370]. While the latter commonly refers to a toolkit of features provided by programming languages. Such built-in features, most of the time, play a decisive role in practitioners' selection of a specific programming language. Among the various programming languages that provide a number of built-in functions, one of the most appreciated by developers is PYTHON that "by design" offers complex built-in functions that can be used during the daily activities of maintenance and evolution [350].

Although the Software engineering (SE) community has spent effort analyzing code quality by considering practices that facilitate software evolution and maintenance, little has been done to look at code quality variation over time. However, this aspect is paramount for practitioners who perform maintenance and evolutionary tasks daily, and they are interested in comprehending how these practices affect code quality attributes.

These uncertainties make releasing automatic *quality assurance tools* and *continuous code quality assurance* tools tricky for the software engineering community, which, due to these limitations, is not able to identify what are the possible factors that developers need to monitor to facilitate the introduction of new features within software system or to avoid the system collapse due to the unrestrained growth of specific code attribute (e. g., Lines of Code (LOC) or coupling) during the system maintenance.

This thesis addresses a significant gap by exploring code quality from an evolutionary perspective. Its *ultimate goal* is to advance our understanding of *Evolutionary Code Quality* (ECQ), namely, the quality attributes that practitioners must monitor to facilitate software maintenance and evolution effectively.

1.2 RESEARCH STATEMENT

Over the last decades, the software engineering community has investigated the topic discussed in this thesis with considerable attention. Despite the high-quality contributions in the code quality field, unexplored aspects are still worthy of investigation.

We resume the main limitations and the critical aspects identified.

REUSE AND CODE QUALITY IN SOFTWARE SYSTEMS. Several investigations have been done to comprehend the relationship between reusability mechanisms and code quality. The main aspect that leads researchers to investigate this relationship can be found in the nature of software systems, generally built using object-oriented paradigms e. g., Java [59, 229, 272], that offers, by design, mechanisms that encourage developers to reuse source code [146]. The main investigations that have been done are related to the relationship between *inheritance* (i. e., the mechanism that permits the inherit attributes and methods from a superclass A to a subclass B), and *delegation* (i. e., the mechanism that permits a class A to use a method of a class B by declaring an instance of B, that will be in charge of actually acting), and their impact on code quality.

Considering previous studies, we identified several critical research gaps (RGaps):

RGap$_1$: ⚠ Previous studies do not focus on evolutionary aspects;

RGap$_2$: ⚠ Previous studies primarily focus on inheritance and delegation without considering other reusability mechanisms;

RGap$_3$: ⚠ Previous studies focus on Java systems without considering how code quality varies in other programming languages.

These aspects are paramount for three reasons: I) the decrease of code quality attributes typically affects maintenance and evolutionary activities [351], and for this reason, it is necessary to consider this aspect to understand possible implications and to estimate their impact on the software system, II) the software engineering community proposes several further reusability mechanisms e. g., design patterns, that need to be explored to comprehend their relationship with code quality attributes over time, III) other programming languages are nowadays considered by practitioners to build evolving systems e. g., Python, and its diffusion into the market need to be explored.

The ultimate goal of this dissertation is to perform a step forward on the so-called "*Evolutionary Code Quality*" i. e., analyze the main quality attributes that practitioners should monitor while evolving or maintaining software systems. The *purpose* is to offer a conceptual framework that software engineers can consult to comprehend how code quality is declining and what metrics need to be monitored during the maintenance and evolution of these systems. *The perspective* is for both practitioners and researchers. The former are interested in comprehending the best practices to adopt during the system maintenance and evolution. The latter are interested in investigating the main factors in software systems with high code quality in multiple contexts. Based on the research gaps above, we identified the main research objective of this thesis:

Objective

Comprehend how code quality varies in software systems and how code metrics affect it.

This thesis recognizes the identified research gaps and the objective and addresses them by defining a high-level research goal (RG):

🔍 **RG.** *Understanding How Reusability Mechanisms and Built-in Features Affect Code Quality Over Time*

The empirical experiments conducted in this thesis adhere to the best practices established in Empirical Software Engineering research. Specifically, the thesis utilizes the *Goal-Question-Metric* paradigm [53] to identify and evaluate products, processes, and resources.

For each study, the forthcoming sections will detail the objectives, research questions, and the instruments utilized to conduct the analysis. To ensure rigour in our investigations, we adhered to the *ACM/SIGSOFT Empirical Standards* [290].

Most of the presented studies involve large-scale mining of software repositories (MSR) by using state-of-the-art tools (e. g., PYDRILLER) to extract information on the project's history (e. g., number of commits, source code metrics, and so on). The sampling of projects and the data collection are performed mainly by analyzing GitHub projects with custom scripts (e. g., Python or Bash scripts) or building ad-hoc tools able to extract specific project characteristics (e. g., number of use of inheritance or delegation mechanism) or already released tools (e. g., the design pattern detector released by Tsantalis *et al.* [346]).

Figure 1.1 provides a generic overview of the research process applied to perform our investigations.

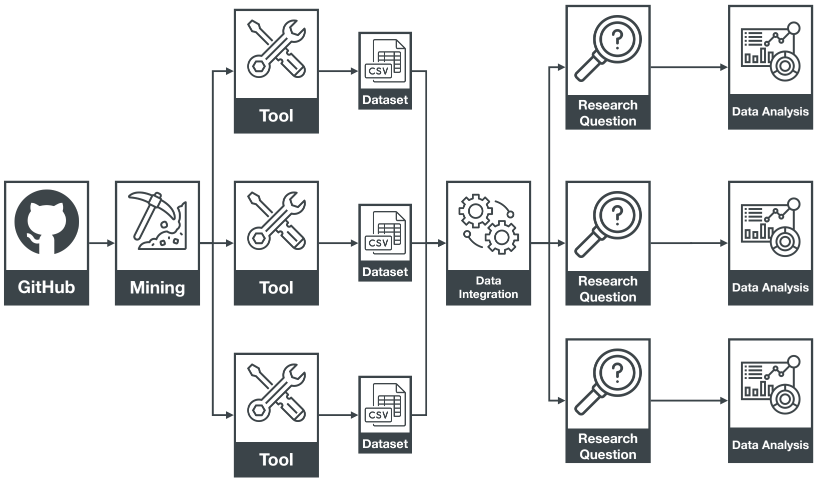

Figure 1.1: Research Method Overview

Starting from the left side of the figure, in all studies, we identified the most appropriate source code platform (e. g., GitHub or SourceForge). Then we cloned repositories and extracted data using multiple tools (e. g., PyDriller, or Decor), then we aggregated data according to the research question that we wanted to answer. Any collected data useful for the experiments was analyzed by using statistical methods. Specifically, we used statistical description to assess basic information about the phenomena under analysis; then, we selected the most appropriate statistical test to analyze correlation aspects between variables according to the specific problem. For instance, we used the regression model to assess the correlation between the adoption of inheritance and delegation with code smells.

Table 1.1: The research studies discussed in this thesis and how they contributed to addressing the problems here faced.

ID	Study Name	Contribution	RG	Ref.
C01	On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality	An empirical analysis of how Inheritance and Delegation evolve over time and their impact on Code Smells	**Main**	Chapter 3
J03	On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort	An empirical analysis on the evolution of Inheritance and Delegation over time and their impact on Defect Proneness and Maintenance effort	**Main**	Chapter 4
C04	The Yin and Yang of Software Quality: On the Relationship between Design Patterns and Code Smells	A large-scale empirical analysis on how design patterns are related on the emergence of code smells	**Main**	Chapter 5
C05	Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Software	A large-scale empirical analysis on the diffusion of code smells over time in AI-enabled Systems and the activities that lead developers to introduce code smells in their systems	**Main**	Chapter 6
J01	On the Use of Artificial Intelligence to Deal with Privacy in IoT Systems: A Systematic Literature Review	A Systematic Literature Review on the use of artificial intelligence to deal with privacy in IoT environments	**Further**	Chapter 8
C02	A Conceptualization and Analysis on Automated Static Analysis Tools for Vulnerability Detection in Android Apps	A large-scale empirical analysis on the capabilities of static analysis tools to detect vulnerabilities in mobile applications	**Further**	Chapter 9

All the results were presented by selecting the most appropriate way (e. g., tables or bar plots) and discussed in separate subsections to increase their understandability and readability.

1.3 RESEARCH CONTRIBUTIONS

This thesis is structured into four main parts. The first part provides the necessary background information, setting the foundation for understanding the subsequent sections. The second part presents studies on evolving systems, focusing on well-known code quality issues such as code smells, defect proneness, and maintenance effort. In the third part, we delve into further studies on code quality, specifically examining emerging technologies (e. g., IoT devices and mobile applications) and addressing principal quality issues that could adversely affect these systems, namely security and privacy. The fourth part concludes with discussions, further analysis, additional considerations, and conclusions, encapsulating the insights gathered throughout this thesis.

Table 1.1 offers an overview of the papers discussed in this thesis, highlighting their respective contributions. The table provides a unique identifier ("*C*" for studies published in international conferences, and "*J*" for those published in international journals), the name of the study, the scientific contribution (i. e., , the results achieved), the research goal ("Main" if the study addresses the primary research goal identified earlier, and "Further" if it pertains to additional studies), and the specific chapter where the paper is discussed.

1.3.1 *Research Contribution on in Evolving Systems*

We addressed the **RG** by performing four empirical studies.

First, we investigated the relationship between inheritance and delegation and their impact on code smells. We selected three Java systems and analyzed them from an evolutionary standpoint. This contribution is presented in publication *C01*. Secondly, we analyzed how inheritance and delegation are related to defect proneness and maintenance effort. We considered 12 Java projects and over 44k commits. The contribution of this study is presented in the publication *J03*.

In addition, we performed an empirical investigation to understand how design patterns are related to code smells over time. The results of this study are presented in *C04*.

Finally, we analyzed the prevalence and activities that induce developers to introduce code smells in systems that extensively use built-in features. In particular, since Python is one of the programming languages that makes the most use of built-in features and simulates itself very well to building AI-enabled systems, we analyzed 200 AI-enabled systems and over 10k releases for our study. We presented the contribution in *C05*.

1.3.2 *Furhter Contribution on Code Quality*

We conducted a Systematic Literature Review (SLR) to comprehend how artificial intelligence handles privacy in Internet of Things (IoT) systems and if the proposed AI techniques can be used in practice to discover or mitigate possible privacy issues. Specifically, we identified 2,202 primary studies by questioning aspects related to the techniques used to identify privacy concerns in IoT systems, the datasets used, the domains where these techniques are adopted, and the tasks performed. The full contribution is presented in *J01*. From the empirical side, we conducted a large-scale empirical analysis of over 6,500 mobile apps using three vulnerability detector tools. The results of the study are presented in *C02*.

We decided to perform these two additional studies by considering code quality attributes related to security and privacy aspects due to the intrinsic characteristics of those two family systems, which in a non-negligible number of cases generate a plethora of data that needs to be exchanged and storage using secure protocols to avoid possible data leakage that could negatively impact the perception of user about the quality of the system [139]. Specifically, the literature on the IoT domains indicates that artificial intelligence techniques lend themselves well to preserving or detecting possible privacy concerns [28]. Due to these considerations, we conducted an SLR to assess previous results and to comprehend the state-of-the-art. In addition, due to the diffusion on ANDROID OS worldwide and the relatively low cost of these devices, we decided to include an investigation to comprehend if the current static-analysis tools can be used in practice to discover vulnerabilities.

1.3.3 *Publicly Available Tools and Replication Packages*

All the raw data, tools, datasets, and scripts are publicly available for each study by including a replication package. Specifically, the online appendixes are available at [119, 411, 412] for Chapters 3 to 6, and [117, 259, 371] from Chapters 8 to 9.

2 BACKGROUND AND RELATED WORK

This Chapter provides the necessary background information helpful to understand the next chapters and summarizes the state-of-the-art giving an overview of closely connecting studies on reusability mechanisms and code quality in software systems.

2.1 KEY CONCEPTS AND DEFINITIONS

The following section provides the principal definitions and elements useful to target our empirical investigation.

2.1.1 *Reuse through Inheritance and Delegation*

In JAVA, a hierarchical dependency between two classes is established by means of two main constructs:

“`extends`”. Through the use of the keyword “`extends`”, a class `A` inherits its state and behavior from a class `B`, establishing a subclass-superclass relation. The attributes defined and the methods implemented in `B` become available when calling objects of the class `A`.

“`implements`”. The adoption of this keyword allows a class `A` to inherit the methods defined within an interface `B`: in particular, a Java *interface* only specifies the blueprint of a class, i. e., the methods that all the classes inheriting from it must provide, without providing a concrete implementation—at least in the older versions. In turn, the inheriting classes must override the acquired methods to specify their behaviour.

These constructs enable the definition of reusability in terms of specification inheritance, implementation inheritance, and delegation [49].

The former represents the possibility to replace one object with another, combining two principles:

- **The Liskov substitution principle**, according to which if an object of type `A` can be replaced anywhere one expects a type `B` object, then `A` is a subtype of `B` [206];

- **Strict inheritance** refers to a subclass `A` that inherits, without any modifications, all the behaviors and properties defined in its parent classes [49]. This form of inheritance ensures that the subclass is a complete superset of the parent class, maintaining all functionalities and characteristics as originally defined.

Implementation inheritance involves a subclass reusing code from a parent class, as noted by [49]. By default, the subclass inherits all operations from the superclass but retains the ability to override some or all of these operations, substituting the superclass's implementation with its own. This mechanism not only facilitates code reuse but also promotes modularity and the ability to extend existing functionalities. The implementation inheritance, however, violates the encapsulation principle [49]: indeed, it does not prevent other client classes to have a direct access to the methods of the superclass, possibly causing clients to invoke those methods improperly. An encapsulation-preserving alternative to implementation inheritance is called *delegation*. This is the mechanism through which a class can delegate an operation to another class without establishing any inheritance relation.

To summarize, specification inheritance, implementation inheritance, and delegation enable the reuse of portions of code in different manners. On the one hand, the reuse expressed in terms of implementation inheritance and delegation exploits the concept of superclasses. On the other hand, specification inheritance is about interfaces.

2.1.2 *An overview of Code Smells*

At the end of the 1990s, Beck and Fowler [104] presented an informal catalog of 22 symptoms of possible design issues, looking at the best and worst software developing practices, coining the term "*code smells*". They discovered that the presence of code smells negatively affects software maintenance and evolution. These smells range from little portions of code e. g., "*Duplication Code*" or "*Long Method*" to entire classes e. g., "*God Class*". Their presence can decrease code quality attributes from multiple viewpoints e. g., increasing the defect proneness [398], or decreasing the code comprehension [147].

Several studies have been conducted to analyze code smells with different objectives. Tufano *et al.* [347] investigated *when and why* developers introduce code smells in software systems and discovered that, in most cases, code smells are removed due to the removal of the files. In other cases, researchers proposed tools useful to discover code smells that range from static analysis tools, e. g., ADoctor [267], Decor [244], JSpIRIT [356], to more elaborate detectors, e. g., iPlasma [225] or PMD [21] that use AI techniques to discover smells. In many cases, the effectiveness of these tools to detect smells appears to be contingent on the chosen technique. Nevertheless, the body of SE knowledge indicates that AI-based solutions may not be entirely well-suited for the task. Indeed, the existing data balancing techniques seem to be inadequate for code smell tasks [275]. Consequently, also applying the state-of-the-art technique to balance code smells datasets, the performances achieved by AI techniques are inaccurate in terms of *Accuracy, F1-score,* and A*UC-ROC* [275].

The main characteristic that unites most previous work is that these studies focus on Java code [268, 323, 347]. If, on the one hand, Java is considered one of the most adopted programming languages to build software systems, on the other hand, researchers noticed a *paradigm shift* [125] by practitioners that tend to select flexible programming languages able to combine multiple paradigms (e. g., Functional Programming and Object-Oriented) and with an extensive set of API enabled the use of AI algorithms during the development of complex systems (e. g., Python) [372]. This paradigm shift enables the possibility of proliferation of specific code smells "*programming languages dependent*", i. e., code smells that emerge in specific programming languages due to their unique characteristics.

2.1.3 *Python-Specific Code Smells*

Python developers underline substantial differences between its programming language and others, not only in terms of syntax but principally in terms of *mindset*—i. e., they do not limit to "translating" from other programming languages to Python—but instead, change the development approaches by adopting unique paradigms that better fit Python philosophy [400]. These differences are so substantial that the Python community coined the term "Pythonic way"[1] to refer to the practice of writing code snippets that enable unique constructs and features provided by Python.

Due to the considerations mentioned above, it is necessary to redefine code smells more specifically to align with the unique characteristics of the language in question. A prime example is *Complex List Comprehension*. This particular code smell occurs in Python list comprehensions that include multiple, intricate expressions. While Python allows for expressions and operations to be applied compactly to each element, overly elaborate expressions can compromise readability and maintainability. This complexity can lead to decreased understandability and potentially increase the likelihood of defects.

Listing 2.1 shows an example of a Complex List Comprehension.

```python
numbers_str = ["24", "15", "21", "27", "35", "40", "45", "50",
    "121", "363"]
filtered_numbers = [int(num) ** 2 for num in numbers_str if
    (int(num) % 3 == 0 and len(num) >= 2 and "5" not in num
    and str(int(num) ** 2) == str(int(num) ** 2)[::-1])]
```

Listing 2.1: Example of Complex List Comprehension.

[1] https://medium.com/swlh/the-pythonic-way-6ad73abfbb00

The code above-mentioned makes the following operations for each “num” in “*numbers_str*”:

- Convert the num from string to integer;
    1. Checks whether num is a multiple of 3;
    2. Checks whether num has more than 2 digits;
    3. Checks whether the digit 5 is not in the original string;
    4. Checks whether the square of num is palindrome.
- If all the controls are successful, the square of num is added in the list ”filtered_numbers”.

To mitigate this smell, developers should consider replacing a complex list comprehension with a loop when possible.

Listing 2.2 shows a refactoring strategy of the previous code.

```
numbers_str = ["24", "15", "21", "27", "35", "40", "45", "50",
    "121", "363"]
filtered_numbers = []
for num in numbers_str:
    num_int = int(num)
    if num_int % 3 == 0:
        if len(num) >= 2:
            if "5" not in num:
                square = num_int ** 2
                if str(square) == str(square)[::-1]:
                    filtered_numbers.append(square)
```

Listing 2.2: Example of Complex List Comprehension refactorized.

2.2 RELATED WORK

This section summarizes the existing studies on code quality. More in detail, in section 2.2.1 will be presented a literature review on how inheritance and delegation are related to code quality; while, in section 2.2.2 will be shown the closest studies on the relationship between design patterns and code smells; lastly, section 2.2.3 will explain the principal differences between previous work and this thesis.

2.2.1 *Literature Review on Inheritance and Delegation on Code Quality in Software Systems*

Software reusability is a key software engineering principle, as it allows developers to reuse pieces of code that have been previously developed and tested [44]. The research community identified some benefits deriving from general features provided by object-oriented programming languages [91, 170, 171, 335]. In this thesis, we review previous literature that has demonstrated the benefits of various reusability mechanisms, as well as their potential drawbacks.

Prechelt *et al.* [281] defined two controlled experiments to verify the relation between inheritance and maintenance effort, showing that keeping the inheritance depth small reduces the overall effort spent by developers while maintaining source code. These results are in line with those reported by Daly *et al.* [78], who conducted a series of controlled studies to investigate the impact of inheritance on source code maintainability. Their results indicated that the higher the depth of the inheritance tree of classes, the lower the ability of developers to maintain those classes. Albalooshi [8] corroborated these findings by showing that multiple inheritance in JAVA may result in undesirable effects on the produced software such as increased coupling, lack of cohesion, and increased software complexity; the author concluded that an improper use of inheritance might lead to major negative effects on source code reusability. Later on, Albalooshi and Mahmood [9] evaluated the implementation of the multiple inheritance mechanism on the reusability of three programming languages like JAVA, PYTHON, and C++, showing that the JAVA programming features lead developers to deteriorate source code quality—as measured by means of the Chidamber & Kemerer (CK) metrics.

Goel and Bathia [122] analyzed whether multilevel inheritance impacts on reusability, conducting an empirical experiment on three C++ systems and finding a negative effect of inheritance on maintainability.

While the studies discussed so far assessed inheritance properties by means of controlled studies, other researchers operationalized source code maintainability in terms of quantitative measurements. For instance, Chawla and Nath [63] assessed that the use of inheritance can have beneficial effects on coupling metrics. Similar conclusions were drawn by Chhikara *et al.* [67], who conducted a larger experimentation on the effects of inheritance on multiple CK metrics. Also Vinobha *et al.* [357] found inheritance to be associated to a higher reusability and maintainability. The three papers just discussed somehow contrasted the findings achieved by the researchers adopting controlled experiments to assess the role of inheritance on software quality, indicating the lack of a clear result on its usefulness. However, when considering the studies proposing quantitative assessments, not all of them found inheritance positive. This is the case of researchers that investigated the relation between inheritance metrics and fault-proneness. A number of papers [2, 38, 84, 317, 390] revealed that the high-values of inheritance metrics, which correspond to larger use of the inheritance mechanism, lead source code to be more fault-prone and might therefore be used as defect predictors. Similar conclusions were drawn when considering change-proneness [56, 211, 357, 408].

A relevant research area is one of code smell detection and refactoring. In this respect, Fowler [105] identified sub-optimal uses of reusability mechanisms and defined code smells like (1) *Refused Bequest*, i. e., subclasses that override most of the methods inherited by the superclass, (2) *Parallel Inheritance*, i. e., inheritance hierarchies that grow too much over time, and (3) *Middle Man*, i. e., classes that excessively use delegation. Ligu *et al.* [202] proposed a dynamic *Refused Bequest* detection approach, while Palomba *et al.* [266] exploited mining software repository techniques for detecting *Parallel Inheritance* instances. Still in terms of code smell research, a number of empirical studies focused on inheritance and delegation. They investigated the diffusion of reusability-specific code smells [172, 265] as well as their impact on defect prediction performance [76, 269].

2.2.2 *Design Patterns and Code Quality in Software Systems*

Design patterns have been introduced by the *Gang of Four* in 1995 [107] and, since then, have been praised as the *Holy Grail* of software reusability. They consist in ready-to-apply solutions to recurring problems in software development, and can aid developers in the design and implementation of source code adhering to good cohesion and coupling principles of object-oriented programming.

Previous work has argued that design patterns may be beneficial for the overall quality of the code. Hegedűs *et al.* [145] investigated the connection between design patterns and software maintainability by performing an empirical study on more than 300 revisions of JHotDraw, a well-known Java framework. They estimated the level of maintainability of source code in terms of different quality attributes, such as number of classes, lines of code, and density of code, and found that each introduction of a design pattern instance generated an improvement in the quality of the project.

However, related work has also discussed that design patterns are not always beneficial for guaranteeing maintainability, especially in terms of understandability and modifiability of the code. Vokáč *et al.* [358] compared the maintainability of programs designed with and without design patterns, by performing a controlled experiment with 44 professionals. They asked participants to execute a number of maintenance tasks on two versions of C++ programs, *i.e.*, one implemented with design patterns and one without. They evaluated the correctness of the executed tasks and the time required by developers, assessing the positive or negative impact of design patterns on software maintainability. They argued that each design pattern has its own nature and proper place of use; they cannot be classified as *good* or *bad* in general terms, but training sessions can improve both the speed and quality of maintenance activities. More recently, Khomh and Guéhéneuc [173] suggested that design patterns may not always have a positive impact on code quality as seen from practitioners' perspective. By performing a survey study with 20 developers, they assessed the perceived impact that design patterns have on the understandability of code. They found that design patterns do not always impact quality attributes positively, as the participants considered that, although they are useful to solve design problems, they often decrease simplicity, learnability, and understandability of the software.

The only work investigating the connection between design patterns and code smells was conducted by Walter and Alkhaeir in 2015 [361]. They selected two medium-size JAVA projects and considered 10 design patterns and seven code smells related to maintainability, *e. g., Feature Envy*, occurring when a method calls methods on another class more times than on the source class, and *Message Chains*, consisting in a client requesting another object, which requests yet another one, and so on, navigating the class structure. Their findings revealed that the presence of design pattern was positively correlated with the presence of code smell, *e. g.,* , the *Proxy* design pattern sometimes led to the introduction of a *Middle Man* code smell.

2.2.3 *Our Contribution on Code Quality*

Our contribution in the field of code quality is complementary to the state-of-the-art. Compared to previous work in this domain, this thesis performs an additional step by considering quality attributes from an evolutionary viewpoint. We addressed code quality in software systems by considering a set of empirical experiments that consider evolutionary aspects, to comprehend how reusability mechanisms and programming language built-in features are related to code quality. More precisely, we conducted experiments considering the impact of reusability mechanisms (particularly focused on inheritance, delegation, and design patterns) on code quality-related aspects (i. e., code smells, defect proneness, and maintenance effort) in evolving systems, and an empirical study on Python projects on the diffusion of code smells and the motivations that lead practitioners to introduce code smells in their systems. Due to the widespread use of Python for developing AI-enabled systems—that is, systems that incorporate at least one AI component—we chose these systems as the subjects for our experiments.

PART II

EVOLUTIONARY PERSPECTIVE OF CODE QUALITY IN SOFTWARE SYSTEMS

3 ON THE EVOLUTION OF INHERITANCE AND DELEGATION MECHANISMS AND THEIR IMPACT ON CODE QUALITY

3.1 INTRODUCTION

Software reusability refers to the development practice of using existing code when implementing new functionalities [44, 326]. This is widely considered a best practice, as it saves developers time, energy, and maintenance costs by relying on previously tested source code on source code previously tested [183, 316].

Contemporary Object-Oriented (O.O.) programming languages, e. g., JAVA, provide developers with various mechanisms supporting code reusability: examples are design patterns [80, 107], the use of third-party libraries [296, 392], and programming abstractions [324]. These latter, in particular, have caught the attention of researchers since the rise of object orientation and were found to be a valuable element in increasing software quality and reusability [180, 239, 297, 302, 388].

Focusing on JAVA, there are two well-known abstraction mechanisms such as *inheritance* and *delegation* [32] (see Chapter 2). These mechanisms are fundamental in Java programming, enabling developers to design more flexible and maintainable software systems.

Still, from an empirical standpoint, a number of studies targeted the role of inheritance and delegation mechanisms for monitoring software quality. Researchers devoted effort on understanding the potential impact of those mechanisms on software metrics [2, 63, 67], maintainability effort and costs [9, 78, 122, 281], design patterns [19, 153], change-proneness [56, 211, 357, 408], and source code defectiveness [38, 84, 317, 390].

While the current body of knowledge provides compelling evidence of the value of reusability mechanisms for the analysis of source code quality properties, we can still identify a noticeable research gap: as Mens and Demeyer [237] already reported in the early 2000s, the *long-term* evolution of source code quality metrics might provide a different perspective of the nature of a software project, possibly revealing complementary or even contrasting findings with respect to the studies that investigated code metrics in a fixed point of software evolution. To the best of our knowledge, Nasseri *et al.* [255] were the only researchers studying the evolution of reusability metrics. They specifically focused on the size of the inheritance hierarchies and aimed at assessing whether developers had the tendency of adding classes at different levels of the hierarchy while evolving their projects: the results reported that the growth of inheritance hierarchies is limited and typically involves up to two levels.

This chapter builds on this line of research by proposing an empirical analysis of how inheritance and delegation mechanisms evolve and their effects on software quality evolution.

Our interest in inheritance and delegation is due to our willingness to (1) investigate built-in abstraction mechanisms that developers are supposed to use to increase the reusability of source code frequently and (2) bridge the gap left by previous studies, i. e., an empirical understanding of the evolutionary aspects of inheritance and delegation might provide a more comprehensive understanding on the role of those mechanisms for source code quality. Our study is conducted on JAVA: while recognizing that other languages (e. g., PYTHON) are becoming more popular, JAVA is still ranked in the top three of the programming languages, according to the TIOBE index.[1] In addition, the structure of the programming language enables a more natural use of inheritance and delegation with respect to other languages [74, 337], which allows us to understand better how these mechanisms evolve and influence code quality. Finally, previous studies investigated JAVA; therefore, our focus enables a comparison with them.

[1] Programming language ranking - Year 2021: https://www.tiobe.com/tiobe-index/.

More particularly, we first mine evolutionary data pertaining to 15 releases of three open-source projects. Then, we statistically compare the number of inheritance and delegation mechanisms implemented over subsequent releases in order to assess the trend followed by the adoption of those metrics. Finally, we build a statistical model relating inheritance and delegation metrics, as well as other confounding factors, to the variation of code smell severity in an effort to understand the impact of reusability metrics on the likelihood of code smells becoming more/less severe over time. The key results of our study report that the adoption of reusability mechanisms increases over time. Yet, when controlled for size, the increase does not appear as statistically significant. In any case, the evolution of inheritance and delegation is statistically connected to the decrease of code smell severity in most cases, and we discovered negative effects only in a few cases. To sum up, this chapter presents the following contributions:

1. Insights into the evolution of inheritance and delegation adoption in open-source systems, which researchers might exploit to understand the developer's code quality practices further;

2. An empirical, evolutionary exploration of the impact of inheritance and delegation mechanisms on code smell severity, which can be of interest to researchers working in the field of code smell detection and prioritization [103, 276, 355], other than for tool vendors interested in providing developers with better monitoring tools for software quality evolution [159, 352].

3. A publicly available replication package [411] contains data and scripts used to conduct our experimentation.

3.2 RESEARCH QUESTIONS AND METHOD

The *goal* of the empirical study was to assess how inheritance and delegation mechanisms evolve over time and how they impact the severity of

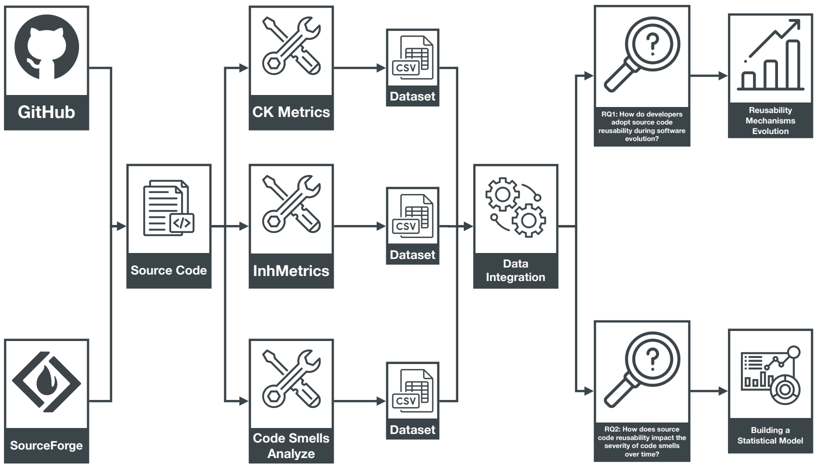
GitHub
SourceForge
Source Code
CK Metrics
InhMetrics
Code Smells Analyze
CSV
Dataset
CSV
Dataset
CSV
Dataset
Data Integration
RQ1: How do developers adopt source code reusability during software evolution?
Reusability Mechanisms Evolution
RQ2: How does source code reusability impact the severity of code smells over time?
Building a Statistical Model

Figure 3.1: Overview of the Method

code smells during software evolution, with the *purpose* of understanding the extent to which reusability mechanisms applied by developers may provide indications on the future quality of source code. The *quality focus* was on the reusability in terms of specification inheritance, implementation inheritance, and delegation and their variability within software projects. The *perspective* was of both researchers and practitioners: the former are interested in gathering a deeper understanding of the role of inheritance and delegation for source code quality, while the latter are interested in better monitoring the code quality, looking at inheritance and delegation metrics. Our analysis was structured around two main research objectives.

We started by analyzing the evolution of inheritance and delegation. An improved understanding of their evolutionary aspects would provide a clearer overview of how developers adopt them in practice.

RQ$_2$. *How do developers adopt source code reusability mechanisms during software evolution?*

We targeted this research question from different angles, each investigating a different code reusability mechanism. This led to the definition of three sub-research questions:

RQ1_1. *How does implementation inheritance vary over time?*

RQ1_2. *How does specification inheritance vary over time?*

RQ1_3. *How does delegation vary over time?*

Once we had assessed the evolution of those mechanisms, we then analysed how such evolution might impact source code quality, as measured by the severity of code smells.

Hence, we asked:

> 🔍 **RQ$_2$.** *How do source code reusability mechanisms impact the severity of code smells over time?*

The empirical study had a statistical connotation: further elaborated later in this section, we approached the research questions by employing statistical tests and models. In terms of reporting, we employed the guidelines by Wohlin *et al.* [376], other than following the *ACM/SIGSOFT Empirical Standards.*[2] The dataset and each script used in order to conduct our study are available in the appendix [411].

Figure 6.1 shows an overview of the methodology followed in order to address our research questions.

Context Selection

The context of the empirical study consisted of 15 releases of three JAVA systems such as JHOTDRAW, APACHE ANT, and JEDIT. Table 3.1 reports

[2]Available at: `https://github.com/acmsigsoft/EmpiricalStandards`. Given the nature of our study and the currently available empirical standards, we followed the *"General Standard"* and *"Data Science"* definitions and guidelines.

information about each of the considered releases, i. e., we provide the name of the system, the release ID, its KLOC, number of classes, and link to the repository. In addition, we also report the number of uses of implementation inheritance (Inh_impl), the number of uses of specification inheritance (Inh_spec), and the number of delegations applied on each of the releases considered (see Section 3.2).

The context selection was driven by one main requirement, namely, our willingness to consider a set of projects that were already selected by similar studies investigating the role of reusability mechanisms [9, 67, 317, 357], in an effort of providing results that might complement the observations done in those previous studies and enlarge our knowledge around the impact of inheritance and delegation on software quality. As such, we identified the three projects more often considered by previous research in the field.

Table 3.1: Descriptive statistics of the considered software systems.

System (ver.ID)	Inh_impl	Inh_spec	Delegation	Classes	KLOC	Link
JHotDeaw 5.2 (v_1)	299	142	57	171	2,275	https://sourceforge.net/projects/jhotdraw/files/JHotDraw/5.2/
JHotDeaw 5.3 (v_2)	398	210	35	241	5,054	https://sourceforge.net/projects/jhotdraw/files/JHotDraw/5.3/
JHotDeaw 6.0 (v_3)	444	183	22	328	10,285	https://sourceforge.net/projects/jhotdraw/files/JHotDraw/JHotDraw60b1/
ANT 1.1 (v_1)	146	13	30	100	5,069	https://github.com/apache/ant/releases/tag/rel%2F1.1
ANT 1.2 (v_2)	188	14	45	166	16,332	https://github.com/apache/ant/releases/tag/rel%2F1.2
ANT 1.3 (v_3)	184	16	46	168	16,841	https://github.com/apache/ant/releases/tag/rel%2F1.3
ANT 1.4 (v_4)	235	53	59	241	39,270	https://github.com/apache/ant/releases/tag/rel%2F1.4
ANT 1.5 (v_5)	327	70	77	397	140,452	https://github.com/apache/ant/releases/tag/rel%2F1.5
ANT 1.6 (v_6)	592	75	81	513	291,882	https://github.com/apache/ant/releases/tag/rel%2F1.6.0
ANT 1.7 (v_7)	702	110	101	734	581,329	https://github.com/apache/ant/releases/tag/rel%2F1.7.0
JEdit 3.2.1 (v_1)	428	136	132	465	83,442	https://sourceforge.net/projects/jedit/files/jedit/3.2.1/
JEdit 4.0 (v_2)	458	134	152	538	116,567	https://sourceforge.net/projects/jedit/files/jedit/4.0/
JEdit 4.1 (v_3)	484	139	183	567	133,491	https://sourceforge.net/projects/jedit/files/jedit/4.1/
JEdit 4.2 (v_4)	506	149	204	701	200,019	https://sourceforge.net/projects/jedit/files/jedit/4.2/
JEdit 431 (v_5)	662	159	209	947	494,462	https://sourceforge.net/projects/jedit/files/jedit/4.3/

Extracting Reusability Metrics

The first step of our empirical study was concerned with the computation of reusability metrics and, specifically, the adoption of specification inheritance, implementation inheritance, and delegation. To extract and quantify the presence of these mechanisms, we developed an *ad-hoc* approach—available in the online appendix [411].

IMPLEMENTATION INHERITANCE (INH_IMPL). For a given type (class), the metric measures the reuse expressed in terms of implementation inheritance. To better explain how to quantify such reuse, consider the following example. Let `A` be a class having `N` methods; let `B` be the superclass of `A` and let's assume that `B` have just one method, named `foo`. To increase the value of implementation inheritance, one of the methods of `A` must invoke `foo`. Our tool mines the source code of two subsequent class releases and checks for cases where the example above appears. It is important to note that if the class `B` is a subclass of another class `C`, all methods of `C` are also considered in the computation.

SPECIFICATION INHERITANCE (INH_SPEC). For a given type (class), the metric measures the reuse expressed in specification inheritance. To quantify the reuse, our devised tool applies the following steps:

- First, it considers all the interfaces. Suppose that the class `A` inherits from two interfaces `B` and `C`, with `C` extending another interface class `E`. In this case, the sum of the interfaces of `A` is 3.
- Second, the concept of strict inheritance must be considered. In the example discussed in the previous point, the tool considers all the extension points of class `A` and verifies that `A` does not override any methods inherited.

DELEGATION (DEL). For a given type (class), the metric measures the reuse expressed in terms of delegation. Given a class `A`, the tool extracts all the instance variables it declares. These represent the input of a "filtering procedure" that filters out the variables that have a basic type (e. g., `int`, `double`, `String`, `boolean`) or have a non-binding type to the considered project, i. e., the variable is of a type coming from an external library, like the class `BottomGroup` of the `javax.swing` framework. This step allows the tool to consider only the instance variables that class `A` uses to call methods of other classes belonging to the same project. The tool verifies whether these remaining instance variables are involved in external calls, i. e., they are used to delegate operations.

It is important to remark that we did not rely on existing metrics, like the Depth of Inheritance Tree (DIT) or the Number of Children (NoC) [68] since we aimed at computing metrics that could have directly expressed the adoption of reusability mechanisms. Indeed, our metrics have a finer-granularity and can indicate the exact constructs added by developers during software evolution, e. g., the inclusion of a new method that delegates its operations rather than a change in the inheritance structure—this would not be possible using existing metrics, as they just provide the result of the actions done by developers, e. g., the increase of the depth of inheritance tree, without indications of how that was obtained.

RQ1. Analyzing the variation of Delegation, Specification Inheritance, and Implementation Inheritance over time

When addressing the first research question, we analyzed the distributions of the three metrics denoting the reuse—*Inh_impl*, *Inh_spec*, and *Del*— to understand how these evolve. For each subsequent release, R_i and R_j, we applied non-parametric statistical tests to verify whether the distribution of each reusability metric differed between R_i and R_j. First, we applied the Mann-Whitney test [235], which is the non-parametric version of the Wilcoxon rank-sum test: the choice was due to the sample size and the non-normality of the distributions considered [73]. Second, we complemented the analysis with the application of the Cliff's Delta (δ) [72], which quantifies the effect size of the observed differences, hence providing a measure of the extent to which the reusability metrics vary over subsequent releases of the considered applications. It is important to note that, when performing the statistical analyses, we normalized the values of the reusability metrics by LOC: this was done to account for the natural evolution that software systems have in terms of size and obtain an unbiased picture of how developers employ the various reusability mechanisms.

The results were intended to be statistically significant at $\alpha = 0.05$. Thus, the null hypotheses tested were:

- $\text{Hn1}_{i,j}$: There is no statistically significant difference between the *Inheritance Implementation* values of version i and *Inheritance Implementation* values of subsequent version j.
- $\text{Hn2}_{i,j}$: There is no statistically significant difference between the *Inheritance Specification* values of version i and *Inheritance Specification* values of subsequent version j.
- $\text{Hn3}_{i,j}$: There is no statistically significant difference between the *Delegation* values of version i and *Delegation* values of subsequent version j.

where $i, j \in \{v_1, v_2, ... v_t\}$, when the system has t versions.

RQ2. Correlation between reuse and code smell severity

We developed a statistical model correlating reusability metrics and other control factors with increased or decreased code smell severity to address the second research question.

RESPONSE VARIABLE. This was represented by the severity of code smells. To compute it, we first selected the actual code smell types investigated. These were:

- *God Class*: A large class with different responsibilities that monopolizes most of the system's processing [105];
- *Spaghetti Code:* A class without structure that declares long methods without parameters [105].
- *Complex Class*: A class poorly understandable and characterized by a high cyclomatic complexity [105];
- *Class Data Should be Private*: A class exposing its attributes, violating the information hiding principle [105];

Two main observations drove the selection of these code smells. Previous studies have established a relation between the reusability mechanisms and source code complexity (e. g., [8, 9]): as such, we selected code smells that are connected to code complexity in different manners, for instance by detecting badly designed, i. e., *God Class* and *Spaghetti Code*, or too complex classes, i. e., *Complex Class*. We also included the *Class Data Should be Private* code smell: its definition suggests a poor use of code reuse principles and, for this reason, we found it interesting to assess the evolution of inheritance and delegation to the erosion of the code connected to this smell.

To detect instances of these code smells we relied on a well-known code smell detector named DECOR [244], still widely used by recent literature [81, 136, 156]. It relies on the computation of code metrics that can capture the properties expressed in the definition of the smells.

For instance, the *Class Data Should Be Private* smell is identified by DECOR as classes that have many variables with visibility `'public'` higher than 10. The detailed detection rules employed as well as the source code of the detector are available in our online appendix [411]. The use of DECOR was motivated by the fact that it represents a good compromise between execution time and detection accuracy—this compromise has been demonstrated multiple times in the past [34, 244, 274]. However, it is worth noting that the use of DECOR did not allow us to focus our study on other relevant code smell types, namely *Parallel Inheritance*, *Middle Man*, and *Refused Bequest*. On the one hand, the former code smell can be detected only using historical analysis [266] and, for this reason, we could not identify it using code metrics. On the other hand, *Middle Man* and *Refused Bequest* have been detected in the past using dynamic analysis [202] but, unfortunately, these approaches are neither publicly available nor easily re-implementable: to avoid the introduction of any bias due to re-implementation, we excluded these smells from our empirical study.

Once we had identified code smell instances, we proceeded with analyzing how their severity evolved over time. To estimate the severity of code smells in the release v_i, we followed the guidelines by Marinescu [226]. In particular, DECOR classifies the presence of a code smell using a heuristic approach that combines multiple metrics: as such, a class is considered smelly *if and only if* a set of conditions are satisfied, where each condition has the form of $metric_i \geq threshold_i$. Hence, the higher the distance between the actual code metric value ($metric_i$) and the fixed threshold ($threshold_i$), the higher the severity of the code smell to that specific metric. Following this reasoning, we measured the severity of code smells as follows: (1) We computed the differences between the actual metric values and the corresponding thresholds used by DECOR [244]; (2) We normalized the obtained differences in the range [0;1] using the min-max strategy [273]; and (3) We computed the final severity score as the mean of the normalized values, ensuring a standardized approach to evaluating the data. This method provides a balanced representation by averaging the individual scores after they have been adjusted to a common scale.

As a final step, for each release pair (v_i, v_{i+1}) of a project P and for each code smell instance cs_j, we computed the difference between the severity of cs_j in v_{i+1} and the one in v_i. If the resulting difference was higher than 0, the severity of cs_j increased: hence, we labeled the event as an *"increase"*; if the difference was negative, then we labeled the case as a *"decrease"*; otherwise, the event was labeled as *"stable"*. These three labels represented the response variable of the four models constructed, i. e., we built one model for each of the code smell considered in the study.

INDEPENDENT VARIABLES. We aimed at assessing the reusability metrics, namely the *Inh_impl*, *Inh_spec*, and *Del* metrics whose definition and computation are reported in Section 3.2.

CONTROL VARIABLES. The variability of code smell severity may depend on different aspects different from the reusability metrics we considered as independent variables.

We accounted for these aspects when modeling our statistical exercise, defining some control variables. We first considered a set of code quality metrics, namely LOC (Lines of Code), WMC (Weighted Methods per Class), RFC (Response for a Class), LCOM (Lack of Cohesion of Methods), CBO (Coupling Between Objects), DIT (Depth of Inheritance Tree), and NoC (Number of Children). We computed these metrics with the MET-

RICS tool.[3] As done for the response variable, we modeled these metrics by considering the difference between their values in the version v_{i+1} and the ones in v_i, i. e., for the sake of consistency, we modeled their evolution and the effect it has on the evolution of code smell severity. It is worth remarking that these control variables are not used by DECOR for the detection of code smells: in other words, there is no dependency between independent and dependent variables—otherwise, this would have caused possible biases when interpreting the statistical results [14].

On the one hand, these metrics have been considered effective to assess source code quality [332, 334]. On the other hand, they estimate a variety of code quality aspects, such as size, code complexity, coupling, cohesion, and propensity to reuse—hence, perfectly fitting our goal of controlling for code quality when evaluating the evolution of code smell severity. Finally, it is worth remarking on the usage of DIT and NoC. These are clearly connected to the reusability metrics we considered as independent variables. Nonetheless, we considered them with the intent of comparing their statistical power to the adoption mechanisms estimated by our metrics: in other words, their employment allowed us to assess the importance of the size of the inheritance tree and the number of children in the inheritance hierarchy with respect to the general usage of inheritance and delegation—note that we assessed the presence of possible multi-collinearity due to these related metrics when performing the statistical modeling.

RUNNING THE STATISTICAL MODEL. Given the nature of our categorical response variable, i. e., the categories *"decrease"*, *"stable"*, and *"increase"*, we used a Multinomial Log-Linear model [340] to study the severity of the four code smells considered. This is a classification method that is applied when the dependent variable is nominal and composed of more than two levels. We built our models using R, exploiting the function `multinom` available in the package `nnet`,[4] i. e., the models were fit via neural networks. When constructing the statistical models, we

[3] https://github.com/qxo/eclipse-metrics-plugin
[4] https://cran.r-project.org/web/packages/nnet/nnet.pdf

considered the problem of multicollinearity, which appears when two or more independent variables are highly correlated and can be predicted one from the other, possibly biasing the interpretation of the results. In the context of our work, we applied the guidelines proposed by Allison [12], who described how to control a model for multicollinearity and when to ignore it. As a result, we did not remove any of the variables. This was because the standard errors of the independent variables were narrow enough not to influence the interpretability of the model negatively: in our case, the standard errors of all the models were lower than 0.9—–note that standard errors must be ≤ 2.5 to produce a sufficiently narrow 95% prediction interval [232].

When interpreting the model results, the multinom coefficients are relative to a reference category and indicate how the independent variables change the chances of the dependent variable being affected with respect to the reference category. We set such a category to *"stable"*: in this way, we could understand how the different independent variables vary, in either a positive or negative manner, the likelihood of the code smell severity being stable over two releases. For example, a negative coefficient for an independent variable of the model built when analyzing the decrease of code smell severity would suggest that for one unit increase of that variable, the chances of variation of the response variable would be increased of the amount indicated the coefficient.

3.3 ANALYSIS OF THE RESULTS

The results of the study are presented in the following.

RQ1. How do developers adopt source code reusability mechanisms during software evolution?

When addressing the evolution of reusability mechanisms, we first analyzed how implementation inheritance, specification inheritance, and

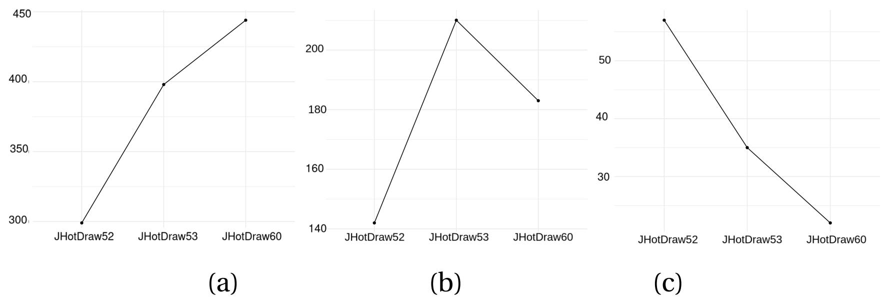

Figure 3.2: How Inh_impl (a), Inh_spec (b), and Delegation (c) change across the considered versions of JHOTDRAW.

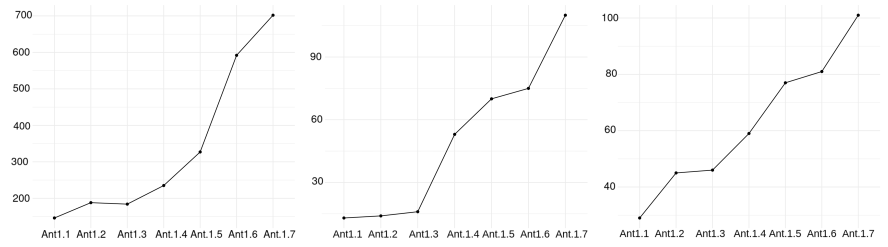

Figure 3.3: How Inh_impl (a), Inh_spec (b), and Delegation (c) change across the considered versions of ANT.

delegation evolved over the considered projects in absolute terms and whether the difference between different releases is statistically significant. Due to space limitation, the detailed table reporting the results of Mann-Whitney and Cliff's Delta is in the online appendix [411]. Figures 3.2 to 3.4 depict the evolution of these mechanisms over the releases of the three considered projects.

The adoption of implementation inheritance follows an increasing trend in three projects. While this seems to suggest that developers use more and more frequently this type of reusability mechanism, we also pointed out the case of JEDIT. In this case, we observed a notable growth between versions `4.2` and `4.3`. To better understand the reason behind this finding, we manually dived into the mailing list of the project, in an effort of understanding whether the developers themselves commented on this aspect in a certain moment in time. This eventually happened on April 10th,

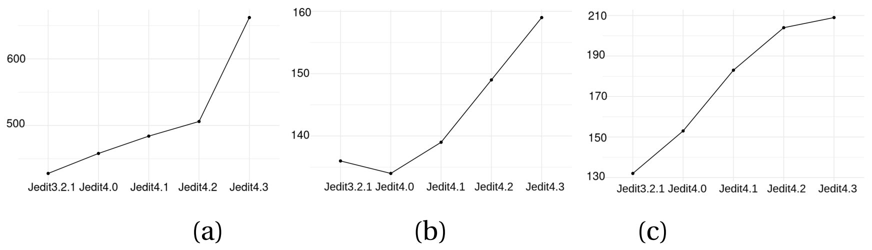

Figure 3.4: How Inh_impl (a), Inh_spec (b), and Delegation (c) change across the considered versions of JEDIT.

2007. As announced in the *"jEdit-announce"* mailing list,[5] the version `4.3` was substantially revised, not only to include new features and bug fixing operations, but also to provide new APIs: these latter modifications have let developers apply consistent editing/refactoring operations of the source code, which implied the improvement of source code design and a higher adoption of inheritance mechanisms. While additional qualitative analyses would be useful to understand the specific reasons why developers increased the use of implementation inheritance while preparing the version `4.3` of the project, our findings suggest that the higher adoption was indeed due to the developer's willingness to provide other reuse mechanisms such as APIs. Nonetheless, when considering the results from a statistical standpoint, we could not identify significant variations—both the Mann-Whitney and Cliff's Delta tests did not reveal a relevant change in implementation inheritance adoption when passing from version `4.2` to version `4.3`. This is likely due to the effect of size, i. e., the ratio of implementation inheritance and size was similar in both releases, even though the reuse was increased in absolute terms. A similar conclusion can be drawn when looking at the statistical tests of the other systems: there were no cases of statistically significant changes between two subsequent releases, meaning that, overall, the use of implementation inheritance does not vary too much over the evolution history when controlled for size.

[5]Thread in the *"jEdit-announce"* mailing list: `https://sourceforge.net/p/jedit/mailman/jedit-announce/?viewmonth=200710`.

Key findings of RQ1$_1$.

implementation inheritance has been increasing over time, although this increase is not statistically significant. In JEdit, the increased adoption was likely due to developers' willingness to improve APIs.

Turning to the specification inheritance, from Figures 3.2-3.4 we could observe a similar trend with respect to the one discussed above. The adoption tends to increase over time in absolute terms, but without any statistically significant change. Hence, we can claim that the evolution is basically stable over time. A slight exception was represented by JHotDraw, where we observed a decrease adoption when passing from release `5.3` to `6`. Analyzing this case further, we could find that the JHotDraw team decided to apply substantial changes to the system, likely affecting the reusability mechanisms previously used and preferring other strategies (e. g., implementation inheritance) over specification inheritance.

Key findings of RQ1$_2$.

The adoption of specification inheritance is stable over time. The only exception was JHotDraw, that preferred to use different reusability mechanisms while defining a new milestone.

Finally, the reuse in terms of delegation follows a similar increasing trend in Ant and JEdit, with the exception of JHotDraw. When analyzing the evolution of the latter system more closely, we could not really derive a clear motivation behind the decreasing trend. This might potentially be connected to a progressive reluctance of designing source code for delegation, however it is important to note that, also in this case, the statistical results did not reveal significant changes. This implies that the differences observed in absolute terms are balanced by the increasing number of lines of code.

Key findings of RQ1$_3$.

The adoption of delegations increases over time, but not in a statistically significant way. The exception is the one of JHotDraw, where the trend observed suggests a progressive reluctance to this mechanism, which might be worth studying in the future.

Table 3.2: Results of the statistical models for each smell in the analysis. The table shows the value of the estimates and the significance through the asterisk.

	Spaghetti Code		God Class		Class Data Should Be Private		Complex Class	
Variables	**Decrease**	**Increase**	**Decrease**	**Increase**	**Decrease**	**Increase**	**Decrease**	**Increase**
Delegation	0.011***	-0.358***	1.054***	-1.363***	0.016***	0.330***	0.011***	-0.358***
Implementation Inheritance	-0.094***	-0.061***	0.048***	0.003***	-0.016***	-0.008***	-0.094***	-0.061***
Specification Inheritance	0.002***	-0.023***	0.028***	0.036***	0.022***	-0.010***	0.002***	-0.023***
DIT	0.060***	0.180***	-0.274***	0.108***	-0.133***	0.004***	0.060***	0.180***
NOC	-0.009***	0.007***	-0.053***	0.002***	0.032***	0.004***	-0.009***	0.007***
LOC	-0.000	-0.000	-0.000	-0.000	0.000**	-0.000	-0.000	-0.000
LCOM	0.910***	-0.101***	-0.177***	-3.218***	0.408***	0.230***	0.910***	-0.101***
WMC	0.0005	-0.0004	-0.002	-0.001	0.001	-0.001	0.0005	-0.0004
CBO	-0.327***	0.201***	-0.679***	0.870***	0.023***	-0.453***	-0.327***	0.201***
RFC	0.001	0.003**	0.008***	0.003	0.002	0.005***	0.001	0.003***

*p<0.1; **p<0.05; ***p<0.01.

RQ2. How do source code reusability mechanisms impact the severity of code smells over time?

Table 3.2 shows the results of the statistical models built for each of the smells considered in Section 3.2. In the first place, it is important to recognize that the decrease of the CK metric values—considered as control variables in our models—correlate well with the decreasing of code smell intensity. This phenomenon was somehow expected since code quality metrics have always been used as variables to predict and monitor code smells [244, 344]: as such, we can confirm the impact of these metrics on the variability of code smells.

Starting from *Spaghetti Code*, we could observe that delegation and the specification inheritance correlate well with the variability of code smell intensity. When they decrease, we found a positive correlation with the increase of the intensity of code smells. This result seems to perfectly fit the nature of this smell. One of the causes leading to the emergence of this smell and its degradation is indeed the inexperience with object-oriented design technologies: our results suggest that a decrease in the use of inheritance and implementation, which are widely considered as relevant for a successful application of object-orientation [105], leads to increase the chances of this smell being harmful. As for the implementation inheritance, we found that its decrease causes instability, i. e., we found negative estimates when considering both the model build to study the increase and decrease of code smell intensity. From a practical perspective, this means that the specific uses of this mechanism may lead to different results.

Moving to *God Class*, we noticed that the increase of delegations may contribute to the decrease of intensity, while both the inheritance metrics create instability. If a class affected by this smell increases the usage of the delegation mechanism, this means that the overall number of responsibilities it manages is reduced: this may explain the reason behind the severity reduction. At the same time, an increasing use of inheritance implies exactly the opposite, with the *God Class* including more and more methods coming from its superclasses. As such, our findings suggest that an appropriate use of delegation might result in an improvement of code quality with respect to the emergence of *God Class* instances: this is also demonstrated by the way some state-of-the-art refactoring tools actually deal with this code smell: as an example, JDEODORANT [101] realizes *Extract Class* refactoring operations by means of delegations, namely by moving methods from the *God Class* to other classes, letting the original class rely on those methods by means of delegation.

As for *Class Data Should Be Private*, we noticed expect for delegation, the lower the presence of inheritance, the higher the chances of the code smell severity being decreased. This result may be concerned with the encapsulation principle that characterizes this code smell. As stated by Gamma *et al.* [108] *"inheritance mechanism often breaks encapsulation, given that inheritance exposes a subclass to the details of its parent's implementation"*. A lower use of these reusability mechanisms naturally makes classes less exposed, hence reducing the risks connected to the presence of a *Class Data Should Be Private*.

Considering *Complex Class*, we noticed that these metrics can help decreasing the variability of the smell. This is, likely, the most interesting outcome of our analysis. Previous studies [8, 9, 304] have established a relation between the use of reusability mechanisms and source code complexity: with respect to those papers, our findings suggest that keeping inheritance and delegation under control may lead developers to reduce the risks of increasing complexity. In this respect, our empirical study provides an additional take on the role of reusability for software maintainability.

Finally, looking at the control factors related to inheritance, i. e., DIT and NOC, we noticed that they lead to higher instability compared to our independent variables, possibly indicating that the pure adoption of inheritance and delegation might be used to better monitoring the variability of code smell severity, correlating better the phenomenon.

Key findings of RQ_2.

In most cases, delegation and inheritance metrics positively correlate to the decrease of the code smell severity. Nonetheless, in some cases, their presence causes instability.

3.4 THREATS TO VALIDITY

This subsection discusses possible threats that could have affected our results and how we mitigated them.

CONSTRUCT VALIDITY. Threats to *construct validity* concern with the relationship between theory and observation. The main discussion point in this respect is related to the dataset exploited. We are aware that the projects selected could have influenced the extent of the analysis, yet we relied on projects that have been previously used in similar experimentations to extract results that might have been as comparable as possible. Future investigations will extend our understanding on the evolutionary aspects of reusability mechanisms. An additional threat concerns the data collection procedures: these were either based on well-established tools, e. g., METRICS, or classical definitions of metrics, e. g., the computation of reusability metrics. In any case, we made all scripts and data publicly available for the sake of verifiability.

As for the identification of code smells, we relied on DECOR [244], which is a state-of-the-art detector [274]. Its accuracy might have influenced the quality of the information reported in the dataset: while we recognize this limitation, we also point out that the choice of this detector was based on the compromise between quality and performance it ensures [34, 244, 274]. In our future research agenda, we plan to assess the impact of false positives/negatives on the achieved results.

INTERNAL VALIDITY. Threats to *internal validity* concern factors that might have influenced our results. In the context of RQ_2, we defined some control variables that could estimate, in a more appropriate manner, the effect of reusability metrics on the variation of code smell severity. The selection of those metrics was based on the state-of-the-art analysis, namely on identifying the metrics that have been previously connected to the evolution of code smells.

CONCLUSION VALIDITY. A major threat to the conclusions drawn is related to the statistical methods employed. In **RQ1**, we used well-known tests widely exploited by the research community, i. e., Mann-Whitney and Cliff's Delta. Before using them, we verified the normality of the data, which is the main requirement leading to their use. As for **RQ2**, the selection of the Multinomial Logistic Linear statistical approach [340] was driven by the fact that our response variable was categorical and composed of three levels. By definition, this statistical approach can handle multiclass problems with categorical and continuous independent variables, therefore fitting the problem of interest. While designing the model, we also controlled for possible multi-collinearity, hence avoiding bias in the interpretation of the results [262].

EXTERNAL VALIDITY. Threats in this category mainly concern with the generalization of results. We analyzed 15 releases of three open-source software systems from different application domains with different characteristics (size, programming languages, number of classes, etc.). Of course, we cannot claim the generalizability of the findings to other systems; our future research agenda includes the extension of the study with a more different set of systems.

4 ON THE ADOPTION AND EFFECTS OF SOURCE CODE REUSE ON DEFECT PRONENESS AND MAINTENANCE EFFORT

4.1 INTRODUCTION

Software reusability is the design principle that allows developers to reuse part of the existing software to implement new features [44, 326]. This practice is widely recognized as one of the key assets of software development, as developers may have multiple benefits, such as the reduction of evolution time, effort, and cost, other than the reduction of risks of source code being affected by defects [183, 306, 316].

Despite the availability of a large body of knowledge on how inheritance and delegation mechanisms contribute to the prediction of source code attributes, most prediction models defined so far made a strong assumption: *developers use reusability principles while evolving source code.*

First, the extent to which these mechanisms are used in practice might notably impact their contribution to prediction models. Second, it is unclear how the relationship between reusability and source code attributes varies and whether inheritance and delegation mechanisms should still be considered for prediction as the system evolves.

In this chapter, we propose an empirical investigation to fill the limitations of current research concerning the adoption of reusability practices and their evolutionary effects on two specific source code attributes such as *defect proneness* and *maintenance effort*. We select these attributes as they represent two interesting use cases to assess reusability mechanisms. On the one hand, these mechanisms are supposed to reduce fault proneness and maintenance effort [183, 306, 316]. On the other hand, several prediction models targeted the early location of defects and estimation of the effort required to perform evolutionary tasks [56, 252, 271].

We first mine the DEFECTS4J dataset to extract commit-level information on the reusability mechanisms adoption. Then, we developed statistical models to assess the contribution of reusability mechanisms on defect proneness—as indicated by the number of defects over time—and maintenance effort—as indicated by the code churn of commits. The main results report on the inheritance and delegation usage patterns of the 12 projects considered, highlighting that (1) developers tend to frequently use these mechanisms and (2) their adoption varies significantly over time. Furthermore, we identify a statistical relation, corroborated by a fine-grained qualitative investigation, between the adoption of inheritance and delegation and both defect-proneness and maintenance effort, hence concluding that software reuse is a relevant component that affects the way source code quality evolves.

4.2 RESEARCH QUESTIONS AND METHOD

The *goal* of the study was to (1) investigate the adoption of reusability mechanisms over time and (2) assess their impact on defect-proneness and maintenance effort. The *purpose* was to understand whether those mechanisms can provide developers with an indication of source code quality variation—considering the defect-proneness and effort to fix faults of a project. The *quality focus* was on the reusability in terms of implementation inheritance, specification inheritance, and delegation and their evolution within software projects. The *perspective* was that of practitioners and researchers: the former are interested in understanding whether the reusability mechanisms can be suitable for monitoring the quality of a system, while the latter are interested in improving their knowledge on how inheritance and delegation mechanisms can vary over time and impact source code quality. The *context* of our investigation was composed of publicly available JAVA projects, as detailed in Section 4.2.

Based on the goal of our study, we formulated three main research questions. The first aimed at understanding the use of source code reusability mechanisms by developers during software evolution.

Specifically, we asked:

🔍 **RQ$_1$.** *How does the use of source code reusability mechanisms vary during software evolution?*

The goal of **RQ$_1$** was that of providing insights on the evolution of reuse mechanisms that might later be exploited to interpret the findings of **RQ$_2$** and **RQ$_3$**. The patterns observed in the context of this research question will also be useful to understand the effects of inheritance and delegation on defect-proneness and maintenance effort, e.g., should we identify an exponential growth in the adoption of delegation, this would potentially make this mechanism more relevant for software evolution, hence influencing more the amount of effort required to apply modifications.

Since we analyze three mechanisms for reusability, i. e., specification inheritance, implementation inheritance, and delegation [49], that can impact software evolution, we considered three sub-research questions:

RQ$_{1.1}$. *How does the use of implementation inheritance vary during software evolution?*

RQ$_{1.2}$. *How does the use of the specification inheritance vary during software evolution?*

RQ$_{1.3}$. *How does the use of delegation vary during software evolution?*

Once the evolution of reusability mechanisms was analyzed, we investigated how the evolution might affect code quality, initially measuring it in terms of fault-proneness. Hence, we asked our second research question:

🔍 **RQ$_2$.** *How do source code reusability mechanisms impact fault-proneness over time?*

Finally, we assessed the impact of reusability mechanisms on the maintenance effort required to fix faults. Among the various *direct* and *indirect* metrics available in literature [377], we operationalize maintenance effort through *code churn*, that is, the number of lines of code modified within a commit. This is an indirect metric that can proxy the actual effort spent by developers when maintaining source code [234, 251, 377].

In particular, we asked:

🔍 **RQ$_3$.** *How do reusability mechanisms impact code churn?*

Figure 6.1 overviews the research process applied to address our research questions. After the first data extraction phase, where we collected data about inheritance, delegation, and other code quality indicators, we integrated the various information for further analysis. This way, the research questions were addressed by employing statistical tests and models (see details in Section 4.2). To design and report the empirical study, we followed the guidelines proposed by Wohlin *et al.* [376] and the *ACM/SIGSOFT Empirical Standards*[1]. We made all the experimental materials (e. g., datasets, scripts) publicly available in an online appendix [119].

Context of the Study

The context of the study was composed of JAVA projects available within the DEFECTS4J dataset, which collects information on over 800 real bugs of open-source systems. According to the official documentation[2] each bug collected into the dataset is characterized by the following properties:

1. It is reported in the issue tracker of the project, has an associated commit message for resolution, and is fixed in a single commit, i. e., the defect resolution never refers to more than one commit;

[1]Available at: `https://github.com/acmsigsoft/EmpiricalStandards`
[2]`https://github.com/rjust/defects4j`

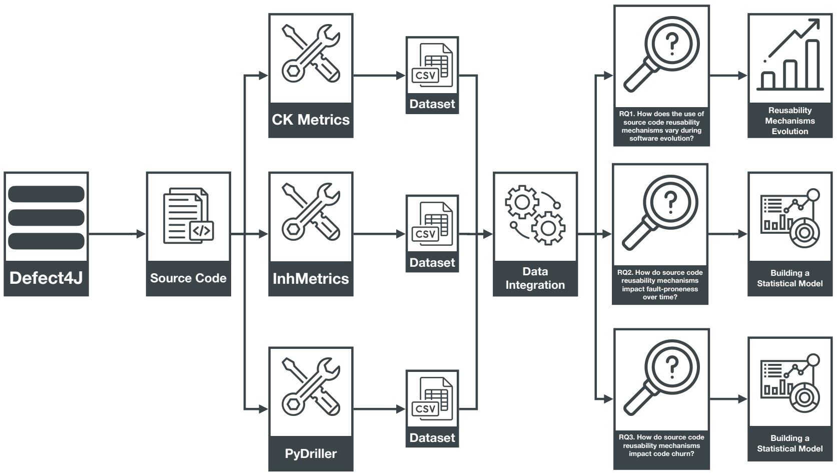
Defect4J
Source Code
CK Metrics
InhMetrics
PyDriller
CSV
Dataset
CSV
Dataset
CSV
Dataset
Data Integration
RQ1. How does the use of source code reusability mechanisms vary during software evolution?
RQ2. How do source code reusability mechanisms impact fault-proneness over time?
RQ3. How do source code reusability mechanisms impact code churn?
Reusability Mechanisms Evolution
Building a Statistical Model
Building a Statistical Model

Figure 4.1: Overview of the research process applied in the study.

2. It is associated with the corresponding triggering test case to facilitate its reproduction;

3. It is minimized, meaning that the DEFECTS4J maintainers manually removed commits that would have induced noise, namely commits that did not actually provide information about the introduction of defects or fixing activity (e. g., commits where refactoring activities were done);

4. The fixing activities modified the source code. This means that the defect introduction can be caused by several factors, e. g., wrong parameters in configuration files and problems in the production class. However, the corresponding fixing only concerns changes within the source code.

By design, the dataset *does not* include all the defects reported in the issue trackers of the considered projects, but only those matching the inclusion criteria reported above. In this respect, there are some considerations to make. First, these criteria led to the definition of a set of defects having two key properties: (1) All the defects were *true positives*, *verifiable*, and *traceable*, meaning that there exists at least one test case letting the defective behavior of the code emerge, other than precise indications on the inducing-fix commit pairs reported by the developers, which were instrumental for our analysis, as further discussed in the following sections; (2) The dataset was carefully designed to avoid potential biases that could arise from *uncontrolled conditions*, such as tangled changes [148], which could significantly impact the validity of our study's conclusions. Specifically, we excluded refactoring actions related to inheritance and delegation that were not associated with defect fixing operations to ensure that the data reflected genuine associations rather than incidental correlations.

As a consequence of these two properties, the choice of DEFECTS4J enabled the investigation of the impact of reuse mechanisms in a *noise-free* environment in which we could have provided more precise insights into the actual role played by inheritance and delegation. In any case, we are aware that the dataset contains a subset of the defects included in the issue trackers of the considered projects and that the missing analysis of some defects might potentially bias our conclusions. In response to this potential threat to validity, we (i) analyzed further the anatomy of the dataset to better characterize our sample - this is discussed in the remainder of this section; and (ii) conducted additional analyses aiming at assessing the types of defects that were not included in our analysis - these are part of Section 8.4.

In addition to the discussion on the use of DEFECTS4J, it is worth remarking that, despite the defects being carefully selected, those defects are of different types and natures, hence representing various defects affecting real-world software systems [322]. Last but not least, DEFECTS4J has been widely used in literature (e.g., [90, 228]), hence representing a valuable asset that enables us to build additional knowledge on a state-of-the-art dataset - this would also be useful for other researchers interested in building on top of our work.

Little has been done to analyze code reuse mechanisms over time and how those may contribute to explaining fault-proneness and maintenance efforts during software evolution. For this reason, our analysis focused on

Table 4.1: Characteristics of the projects considered in the study. The column *'LOC'* provides a range reporting the minimum and maximum values observed over the history of the projects.

Project Name	# Bugs	Pull Request	Contributors	Stars	Forks	Commits	Branches	LOC	Analyzed
Commons-Codec	18	9	40	364	207	2,244	7	48k - 34k	🟢
Commons-Cli	39	8	42	255	154	1,169	4	5k - 16k	🟢
Commons-Collections	4	37	62	551	389	3,729	8	49k - 60k	🟢
Commons-CSV	16	8	37	281	220	1,796	4	166k - 166k	🟢
Commons-Compress	47	9	67	231	210	3,602	9	129k - 91k	🟢
Gson	18	151	125	21,2k	4,1k	1,668	14	68k - 70k	🟢
Jackson-Core	26	2	63	2.1k	690	2,124	21	33k - 66k	🟢
Jackson-Databind	112	19	198	3,1k	1,2k	6,578	22	98k - 235k	🟢
Jackson-Dataformat-XML	6	3	26	497	189	1,318	19	59k - 117k	🟢
Commons-JXPath	22	8	17	18	40	601	4	46k - 26k	🟢
Joda-Time	26	2	77	4,8k	922	2,196	6	103k - 164k	🟢
Closure-Compiler	174	6	472	6,5k	1,1k	17,962	76	60k - 60k	🟢
JSoup	93	43	99	9,6k	2k	1,693	3	39k - 34k	🔴
Commons-Lang	64	92	174	2,3k	176	6,859	8	160k - 190	🔴
Commons-Math	106	68	48	451	71	7,004	17	58k - 63k	🔴
Mockito	38	7	246	13,1k	2,3k	5,787	16	73k - 94k	🔴
JFreeChart	26	22	24	866	355	4218	3	250k - 290k	🔴

the analysis of code reuse mechanisms from a low granularity perspective, i. e., commits. We analyzed over 44,900 commits. With respect to our initial plan [120], we had to discard five projects from the total amount of systems available in the dataset. This was mainly due to repository inconsistencies caused by developers' removal of defective commits.

Table 4.1 reports statistics of the projects included in the DEFECTS4J dataset. For each project, the table provides (I) the number of defects, (II) process metrics such as number of commits, number of pull requests, and number of contributors; (III) its minimum and maximum LOC; and (IV) if the project could have been analyzed. More particularly, we exploited the latest version of DEFECTS4J (v2.0.0). The defects contained in this version were identified by the original authors using JAVA 1.8, which is the JAVA version used by all the projects considered in the study. The reliance on JAVA 1.8 had some implications on the number of defects reported in the dataset. More particularly, some behavioural changes introduced under JAVA 8 did not allow the verification of 29 of the defects reported in previous versions of DEFECTS4J. As such, these 29 defects were considered *deprecated* and *no longer relevant* in DEFECTS4J 2.0.0. In the light of this consideration, we excluded them from our study. These defects indeed violated the first property mentioned above: on the one hand, they were not verifiable; on the other hand, they were not necessarily true positives, as they were re-labelled by the original authors as non-defective when verifying them through the most appropriate JAVA version, namely the one employed within the corresponding systems.

Data Extraction Procedure

To answer our research questions, we quantified the reusability mechanisms employed within the considered software projects. To this aim, we operationalized three metrics capturing reusability mechanisms such as implementation inheritance, specification inheritance, and delegation. We did not rely on existing metrics, like the Depth of Inheritance Tree (DIT) or the Number of Children (NoC) [68], since we aimed at computing metrics that could have *directly* expressed the adoption of reusability mechanisms. Indeed, our metrics have a finer granularity and can indicate the exact constructs added by developers during a change/commit, e.g., the inclusion of a new method that delegates its operations or a change in the inheritance structure—this would not be possible using existing metrics, as they just provide the result of the actions done by developers, e.g., the increase of the depth of inheritance tree, without indications of how that was obtained. To compute the implementation inheritance, specification inheritance, and delegation metrics, we used a tool already validated in our previous work [118]. It computes the metrics following these patterns:

SPECIFICATION INHERITANCE. Given a class *B*, the tool considers the specification inheritance as the arithmetical sum of each interface used by *B*. For instance, suppose that *B* inherits methods from two interfaces *A* and *C*, and *C* in turn inherits methods from another interface *D*. In this case, the specification inheritance for *B* is 3.

IMPLEMENTATION INHERITANCE. Suppose *B* is a sub-class of *A*, the tool considers the implementation inheritance as the arithmetical sum of each method in *A* called by some method in *B*. For example, suppose that *B* is a class with N methods, and *A* a class with just one method call `bar()`. To increase the number of implementation inheritance by one, one of the methods in *B* must invoke `bar()`.

DELEGATION. Given a class *A*, the tool considers the delegation metric as the arithmetical sum of each non-primitive variable (i.e., variables different from `int`, `double`, `String`, and so on) or variables that do not have a binding type provided by external libraries (e.g., `Checkbox` offered by `javax.swing` framework). For each variable, the tool verifies if it is only used to invoke external objects.

The metrics were computed over all the commits of the considered systems and were used to address $\mathbf{RQ}_1$. Specifically, for each commit we computed the sum of (i) specification and implementation inheritance uses and (ii) delegation uses by statically analyzing the files involved in the commit. As for $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$, we collected information on defects and code churn. To this aim, we mainly relied on the information made available by the DEFECTS4J dataset. In particular, for each project of the dataset, DEFECTS4J assigns to each defect a unique ID and stores an inducing-fixing commit pair, i.e., a pair of commits reporting when the defect was introduced and fixed, respectively, over the history of the project. Starting from these inducing-fixing commit pairs, we could reconstruct the defect history of each project by overlaying them on the full set of commits of the project and considering as defective all the commits between the inducing-fixing commit pairs. As for the code churn, these were collected by exploiting PYDRILLER, an automatic static analysis tool that can analyze GIT repositories to extract information about commits, developers, modifications, diffs, and source code.[3] In our case, we run PYDRILLER over the commits of the considered systems and extracted the number of modifications performed by developers, i.e., the code churn.

Data Analysis Procedure

The collected data were further analyzed as follows:

[3] https://pydriller.readthedocs.io/en/latest/intro.html

1. ***RQ*$_1$** *- Evolution of reusability mechanisms over time.*

   To address this research question we analyzed how reusability metrics (implementation inheritance, specification inheritance, and delegation) vary over the evolution of the software systems considered. In particular, we employed basic statistical analysis and visualized results using plots.

2. ***RQ*$_2$** *- Impact on defect-proneness of reuse over time.*

   In this respect, we built a statistical model to verify how reusability metrics impact the variability of defects in the source code.

3. ***RQ*$_3$** *- Impact on maintenance effort of reuse over time.*

   Similarly to **RQ**$_2$, we built a statistical model to verify how reusability metrics impact the maintenance effort to fix a bug.

The statistical models were devised as reported in the following.

INDEPENDENT VARIABLES. We used the reusability metrics, i. e., implementation inheritance, specification inheritance, and delegation, as independent variables.

RESPONSE VARIABLE. In the context of **RQ**$_2$ we were interested in understanding how the reusability metrics impact the defect-proneness of software systems over time. Starting from the defect history built by exploiting DEFECT4J, we modeled our response variable as follows. Let C_i be a generic commit of the change history of the project P. The number of defects affecting P at the time of C_i was computed through the $\#defects(C_i)$ function, which relies on the following system of equations:

$$\begin{cases} \#defects(C_i) = \#defects(D4J_{C_i}) - \#fixedDefects(D4J_{C_i}), \text{ if } i = 1; \\ \#defects(C_i) = \#defects(C_{i-1}) + (\#defects(D4J_{C_i}) - \#fixedDefects(D4J_{C_i})), \text{ if } i > 1; \end{cases} \tag{4.1}$$

where $\#defects(D4J_{C_i})$ indicates the number of defects in DEFECTS4J having as inducing commit C_i, $\#fixedDefects(D4J_{C_i})$ indicates the number of defects fixed in the commit C_i, computed as the amount of defects fixed according to DEFECTS4J in C_i, and $\#defects(C_{i-1})$ indicates the number of defects affecting P at commit C_{i-1}. As shown, we had to distinguish the case of the first commit (i=1) from the rest (i>1). When considering the first commit, there cannot indeed be previous fixing operations that influenced the number of defects and, as such, the number of defects at the first commit is only due to the difference between the number of defects pointed out by DEFECTS4J and the number of defects fixed in the same commit. When considering the other commits, instead, the number of defects at the time of the generic commit C_i is given by the total number of defects at time C_{i-1} plus the operations performed within C_i, both in terms of defects introduced and fixed. After computing the number of defects affecting the considered systems at each commit, we analyzed how this number varied.

Let C_i and C_{i+1} be two subsequent commits of the change history of the project P; we labeled the commit pair (C_i, C_{i+1}) as *stable, increased,* or *decreased* using the $label(C_i, C_{i+1})$ function described in the following:

$$label(C_i, C_{i+1}) = \begin{cases} `Stable' & \text{if} \#defects(C_i) = \#defects(C_{i+1}); \\ `Increased' & \text{if} \#defects(C_i) < \#defects(C_{i+1}); \\ `Decreased' & \text{if} \#defects(C_i) > \#defects(C_{i+1}). \end{cases} \quad (4.2)$$

In other terms, we exploited the information previously collected on the number of defects at each commit of the change history of the project P to describe how the amount of defects varied over time.

In **RQ**$_3$, instead, we were interested in assessing the effect of reusability metrics on the effort required to fix defects, as measured by code churn. Starting from the defect history of each project, we considered, as relevant for the research question, the commits marked as fixing commits.

Afterwards, we computed our response variable as the sum of the code churn of the files involved in those commits.

CONTROL VARIABLES. We calculated several control variables to ensure that the impact on the response variables in our statistical models was not solely due to the independent variables. We computed the Chidamber and Kemerer (CK) metrics [68], which include the following: *DIT* (Depth of Inheritance Tree), *NOC* (Number Of Children), *LOC* (Lines of Code), *LCOM* (Lack of Cohesion of Methods), *WMC* (Weighted Methods per Class), *RFC* (Response for a Class), and *CBO* (Coupling Between Objects). These metrics are crucial for understanding various structural attributes of software that could influence the outcomes of our models.

In $\mathbf{RQ}_2$, we also considered the code churn as control variable as suggested by previous findings in the literature [252], i.e., we verified whether the variation of the number of defects was due to the amount of changes performed by developers within commits. This metric was not considered in $\mathbf{RQ}_3$, as it was directly connected to the response variable and could, therefore, bias the conclusions.

With respect to the control variables considered in the study, it is important to discuss the role of *NOC* and *DIT*. These two metrics are by definition connected to code reusability and measure indeed two aspects related to how developers reuse existing source code through inheritance. We included them with the intent of comparing their statistical power to the reusability metrics considered as independent variables. In other terms, the inclusion of *NOC* and *DIT* allowed us to assess the extent to which the reusability metrics we computed represent relevant factors for the response variables when compared to state-of-the-art metrics.

Before building the statistical models, we assessed the presence of possible multi-collinearity concerns. These arise when two or more variables are excessively correlated, possibly biasing the statistical model and the subsequent interpretation of the results [262]. In this respect, we followed well-established guidelines [12, 201]. For each pair of variables, we computed the Spearman's correlation coefficient [336]. If this scored higher than 0.7, we removed the variable having the most complex definition to favour explainability. For instance, we preferred keeping the *LOC* metric rather than *WMC* to make interpreting the results easier.

CHOOSING THE STATISTICAL MODEL. To address **RQ**$_2$ we built a *Multinomial Log-Linear Model* [340]. This model generalizes logistic regression to multi-class problems, matching our need to have a model able to handle our response variable composed of three values (*"stable"*, *"increased"*, *"decreased"*). As done in our previous work [118], we used R for running the analysis using the function MULTINOM available in the package NNET.[4]

In **RQ**$_3$ we had to build a different model because of the nature of the response variable, i. e., code churn. In particular, we built a *Generalized Linear Model* [95] using the GLM function available in R.

Public Availability of Data

To guarantee the replicability of our work and enable other researchers to build on top of our analyses, we made all data and scripts publicly available in our online appendix [119].

4.3 ANALYSIS OF THE RESULTS

In the following sections, we report and discuss the results addressing the research questions of the empirical study. For the sake of comprehensibility, we split the discussion by **RQ**.

RQ$_1$ *- On the Variation of Reusability Mechanisms in Source Code*

Figure 4.2 shows how the three reusability mechanisms considered in our study, i.e., implementation inheritance, specification inheritance, and delegation, evolve in the considered software projects. Each row of the

[4]https://cran.r-project.org/web/packages/nnet/nnet.pdf

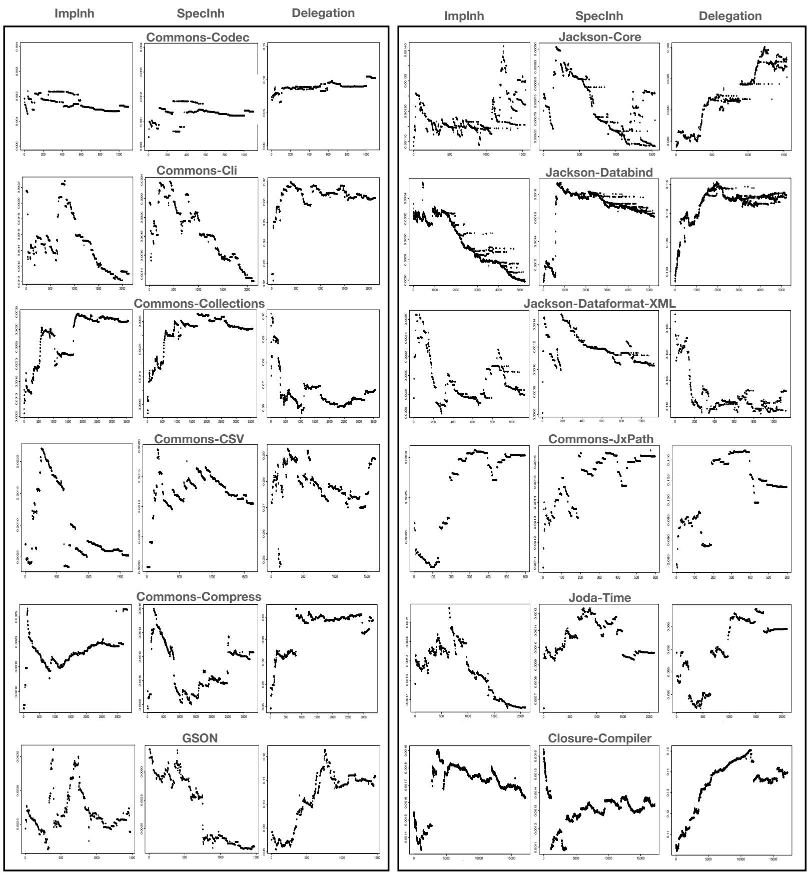

Figure 4.2: RQ$_1$. Adoption of Reusability Mechanisms Over Time.

figure reports the evolution of the metrics for the two projects separately. To facilitate the interpretation of the results and enable a more seamless comparison of evolutionary trends across diverse projects, we normalized the reusability metrics by lines of code—the figure shows the amount of implementation inheritance, specification inheritance, and delegation mechanisms applied per line of code over the evolution history of the considered projects. These trends were used to interpret the results and address the specific sub-research questions defined in the context of **RQ**$_1$.

RQ$_{1.1}$ - *Variation of Implementation Inheritance Over time.*

As for the implementation inheritance, the trends in Figure 4.2 do not always follow a common tendency among the projects.

INCREASING - DECREASING PATTERN. As illustrated in Figure 4.3, we observed an initial increasing trend in the adoption of implementation inheritance in seven projects, namely CLOSURE-COMPILER, COMMONS-CLI, COMMONS-CSV, GSON, JACKSON-DATABIND, JACKSON-DATAFORMAT-XML, and JODA-TIME, which was subsequently followed by a decline in usage.

While the shape of the curves varies from case to case, we can still see a common pattern. When we look at these cases, we can identify a similar behavior among the developers of those systems. In all the cases, the adoption of implementation inheritance quickly increased during the first commits, suggesting that developers approached the design of the systems to take reusability into account. Nonetheless, the trend quickly decreased after a while, leading implementation inheritance to be used less and less over time.

This trend leads us to formulate two observations. Firstly, the decline in adoption following a peak could be indicative of a phenomenon known as “design erosion” in the literature (in cases where the principle of Liskov is violated) [349]. Regardless of the intentions of developers and designers, software design tends to degrade over time due to ongoing changes and increasing complexity, as highlighted by Lehman’s laws

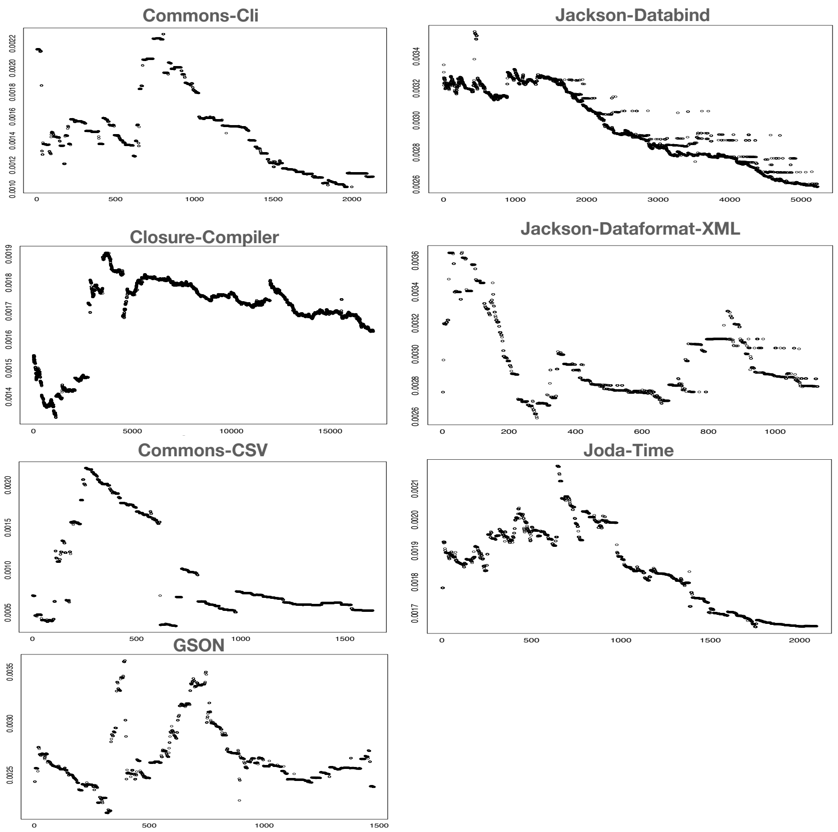

Figure 4.3: Increasing - Decreasing Pattern.

[192]. This erosion can also be attributed to inadequate utilization of software quality measures, as emphasized in previous research [87, 351, 352]. Our findings seem to suggest implementation inheritance is not exempt from this trend, and its adoption is likely to decrease over time.

In the second place, the *"increasing-decreasing"* trend might have implications on how reuse mechanisms should be considered within prediction approaches, e.g., defect prediction. Indeed, the employment of implementation inheritance should be carefully considered, and perhaps the usage trend might even lead to the definition of novel feature selection procedures that monitor how developers use certain programming constructs to inform the model of the most promising features to consider in that evolution moment.

STEADY-INCREASING PATTERN. Looking at Figure 4.2, we can identify three less common usage patterns. In particular, two projects, namely COMMONS-COLLECTIONS (3rd row) and COMMONS-JXPATH (4th row), appear to exhibit a *"steady-increasing"* trend. The nature of these projects seems to offer a natural explanation for this trend. The former project provides a framework to use efficient data structures in JAVA, while the latter implements an interpreter of the XPATH expression language. Both projects are structured so that most of the source code relies on a core set of classes. For instance, in the COMMONS-COLLECTIONS project, classes within the `list` package establish the foundation for creating various advanced element lists. This seems encouraging developers to employ reuse mechanisms like implementation inheritance.

STABLE PATTERN. Two other projects, namely COMMONS-CODEC (1st row) and JACKSON-CORE (1st row) of Figure 4.2, follow a *"stable"* trend. In both cases, the amount of implementation inheritance uses remains constant throughout the evolution. We analyzed the repositories of those projects deeper to understand this trend. While we could not identify any specific tool or verification procedure conducted by developers to keep reusability under control, we could observe that most of the commits performed over the last years were *peripheral* [20], namely, they pertained to packages of the systems other than core. This may explain the observed trend: developers did not modify any central part of those systems, leaving the original design stable and avoiding an excessive effect of design erosion.

DECREASING - INCREASING PATTERN. COMMONS-COMPRESS project (5th row in Figure 4.2) exhibited an anomalous trend which we coined *"decreasing-increasing"*. After a greater adoption of implementation inheritance, the trend steadily decreased before increasing again, but at a lower rate. We manually dived into the repository in search of possible explanations. We discovered that after the release of the second version of the project in 2010 (release `commons-compress-1.1`), the release engineering process of the system changed, passing from annual to monthly releases. This switch caused a substantial rework of the original architecture, replacing existing code with third-party libraries. Consequently, the overall amount of implementation inheritance uses suddenly decreased in favour of other code reuse mechanisms. Afterwards, the developers of the system kept the implementation inheritance under control, leading to an increasing usage trend.

$RQ_{1.2}$ - Variation of Specification Inheritance Over time

When considering the specification inheritance, the usage patterns identified in **$RQ_{1.1}$** still hold. In particular, we observed the same *"increasing-decreasing"* trend in COMMONS-CLI, while in COMMONS-CODEC a *"stable"* trend. These findings seem to suggest the existence of a possible strict (cor)relation between implementation and specification inheritance throughout the evolution of software systems, which might depend on the willingness of developers to take (or not) code reusability into account when evolving source code. Part of our future research agenda will consider the effects of this co-evolution of metrics on software quality.

$RQ_{1.3}$ - Variation of Delegation Over time

Regarding the delegation, we could observe similar usage patterns discussed above. Nonetheless, we could also discover situations where the evolution of delegation followed an opposite trend with respect to implementation and specification inheritance ones. This is, for instance, the case of COMMONS-COLLECTIONS. Indeed, starting from a high adoption during the first development phases, the amount of delegation used kept decreasing till reaching a stable level. This result was, however, somehow expected as inheritance and delegation are alternatives to each other [49] and, therefore, an increasing use of one may lead to a decreasing use of the other. Similar results were observed when analyzing other projects, e.g., CLOSURE-COMPILER JACKSON-CORE and COMPRESS.

The apparent synergy between inheritance and delegation could offer an opportunity for source code quality predictive models. These models could decide which metrics to focus on at different stages of development. In this way, the models could rely on metrics that can best represent the current state of the system under analysis, potentially improving their predictive capabilities.

Key findings of RQ_1.

In 7 out of the 12 analyzed projects, both implementation and specification inheritance exhibited an *"increasing-decreasing"* trend, with design erosion identified as the most probable cause for this pattern. Usage patterns in other projects were less consistent and varied depending on their specific scopes. Additionally, we observed that delegation maintained an orthogonal relationship, showing an opposite trend compared to both implementation and specification inheritance in four of the systems analyzed. The findings from **RQ_1** could have significant implications for predictive software quality analytics. Models within this field could be enhanced by incorporating the identified usage trends, enabling more precise predictions about which metrics are most effective at various stages of software development.

Table 4.2: RQ_2. Variables removed because of multi-collinearity.

Project	Discarded Variables
Commons-Codec	RFC, NOC
Commons Cli	DIT, NOC, InhImp
Commons-Collections	WMC
Commons-CSV	RFC
Commons-Compress	RFC
Gson	RFC
Jackson-Core	WMC, RFC, DIT, InhImp
Jackson-Databind	RFC
Jackson-Dataformat-XML	WMC, RFC, DIT
Commons-JxPath	DIT
Joda-Time	WMC, RFC, DIT
Closure-Compiler	RFC

RQ_2 *- The Impact of Reusability Metrics on Defect-Proneness*

In this sub-section, we report the results when studying the impact of reusability metrics on the defect-proneness of source code.

MULTI-COLLINEARITY ANALYSIS. Before discussing the results of the statistical model, it is worth reporting the outcome of the multi-collinearity analysis—which was performed to make sure that no correlated variables were employed within the statistical model and could bias the interpretation of the results (see Section 4.2). Table 4.2 lists the variables removed after the application of the correlation analysis. In the first place, we found that RFC was the metric most often removed: in all the cases, it was correlated with LOC and, therefore, we preferred keeping LOC because of its highest degree of interpretability. Secondly, in three projects, i.e., COMMONS-COLLECTIONS, JACKSON-CORE, and JODA-TIME, the WMC metric was removed, again for its correlation with LOC. We also discovered correlations between DIT and NOC in two projects such as

COMMONS-CODEC and COMMONS-CLI: we kept NOC, namely the metric reporting the number of immediate subclasses of a class. In the cases of JACKSON-DATAFORMAT-XML and JODA-TIME, we found a correlation between DIT and specification inheritance: as the latter was one of the independent variables, we preferred keeping it. Finally, we identified correlations between specification and implementation inheritance in the projects COMMONS-CLI and JACKSON-CORE—these correlations could be already hypothesized looking at the trends observed in the context of **RQ**$_1$: in these two cases, we were obliged to remove one of the independent variables and decided to opt for implementation inheritance.

STATISTICAL MODEL EXPLANATION. Table 4.3 illustrates the results of the statistical models built in **RQ**$_2$. The independent variables and control variables are reported on the rows, while the various considered systems are reported on the columns—empty cells indicate that a certain variable was removed from the analysis of a specific system as a consequence of the multicollinearity analysis, while the number of observations (the commits analyzed) for each project is reported in the header of each column. The statistical codes report the p-value for each variable and each project and were used to interpret the results obtained. According to Table 4.3, a higher amount of '*' implies a higher statistical relevance of a variable with respect to decrease (↓) or increase (↑) of the likelihood to affect the defect-proneness of source code.

STATISTICAL MODEL ANALYSIS. Looking at the table, various considerations can be drawn. First and foremost, in 10 out of the total 12 projects we found at least one of the inheritance metrics to be a statistically significant factor to explain the defect-proneness of the considered systems. The NOC metric, in particular, is the one being relevant in more systems. On 8 projects the metric was observed to explain both the increase and decrease of defect-proneness.

To comprehend how the metric affects the phenomenon under analysis, we analyzed the sign of the coefficients. The coefficients of a *Multinomial Log-Linear* model relate to a reference category and indicate how the

Table 4.3: RQ$_2$. Results of the statistical model.

	Com.-Codec N=2,134		Com.-Cli N=1,099		Com.-Col. N=3,560		Com.-CSV N=1,634		Comp. N=3,305		Gson N=1,478	
	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
DiffWMC	-10.098	2.280	-0.691	2.416			-3.539	0.627	-5.248***	-1.903	3.261	-1.305
	(7.495)	(10.981)	(3.434)	(3.602)			(2.559)	(5.136)	(0.033)	(4.387)	(30.899)	(25.907)
DiffNOC					-0.038	0.004	-4.413***	1.052***	10.188***	-5.653***	-0.159	1.536***
					(0.050)	(0.013)	(0.156)	(0.176)	(0.002)	(0.051)	(0.275)	(0.415)
DiffLCOM	0.092	0.054	0.166	-0.744***	0.422	0.476	-0.056	-0.040	-0.066	0.046	0.013	-0.159
	(0.140)	(0.261)	(0.256)	(0.244)	(0.808)	(22.615)	(0.066)	(0.130)	(0.335)	(0.125)	(1.242)	(1.157)
DiffDIT	11.927***	−0.183***	0.012	-0.0003	0.012	-0.0003	-4.526***	0.661***	12.511***	-5.151***	0.696	1.896**
	(0.269)	(0.033)	(5.830)	(0.023)	(5.830)	(0.023)	(0.238)	(0.169)	(0.002)	(0.125)	(0.466)	(0.798)
DiffCBO	-5.434	-9.729***	-0.645	-5.947	-0.878	-0.495***	-0.994	-3.484	-4.163***	1.467	-17.977	-3.472
	(5.898)	(0.243	(3.617)	(3.821)	(58.069)	(0.103)	(4.485)	(8.728)	0.021)	(2.717)	(12.462)	(12.553)
DiffRFC	4.123	-0.030	-0.014	4.123	1.013	1.924						
	(5.784)	(1.106)	(1.027)	(5.784)	(6.087)	(14.548)						
DiffLOC	0.005	0.075	0.002	0.056	-0.611	-0.099	0.175	0.045	0.149*	0.024	0.115	0.689
	(0.302)	(0.346)	(0.139)	(0.200)	(1.053)	(11.153)	0.121)	(0.236)	(0.090)	(0.104)	(1.273)	(1.373)
DiffDelegations	0.058	-0.060	0.017	0.001	-0.069	0.004	0.031	-0.058	0.013	-0.003	0.068	-0.018
	(0.049)	(0.077	(0.022)	(0.025	(0.137)	(0.654)	(0.076)	(0.147)	(0.017)	(0.013)	(0.059)	(0.078)
DiffSpecInh	-1.791	-1.510	0.070	1.382***	1.013	1.924	-0.571	-1.495	-0.187	-0.148	-0.356	0.226
	(1.685)	(3.395	(0.618)	(0.542)	(6.087)	(14.548)	(5.267)	(10.178)	(0.862)	(0.637)	(3.632)	(1.865)
DiffimpInh	-0.060	0.046			0.767	0.771	-1.141	-0.094	-0.337	0.154	0.047	0.267
	(0.940)	(2.134			(3.457)	(13.557)	(2.923)	(4.432)	(0.488)	(0.383)	(2.105)	(1.713)
Churns	-0.002	-0.002	-0.002	-0.005	-0.026	-0.100	-0.004	-0.009	-0.003	-0.0003	-0.015	-0.007
	(0.003)	0.004)	(0.002)	(0.004)	(0.038)	(0.127)	(0.007)	(0.013)	(0.002)	(0.001)	(0.014)	(0.009)
Constant	-4.762***	-4.717***	-3.472***	-3.448***	6.429***	-6.230***	4.605***	4.520***	-4.236***	-4.327***	-4.910***	-5.051***
	(0.248)	(0.246)	(0.187)	(0.187)	(0.536)	(0.527)	(0.265)	0.264)	(0.160)	(0.159)	(0.369)	(0.372)

	Jack.-Core N=1,543		Jack.-Datab. N=5,228		Jack.-XML N=1,128		Com.-JxPath N=598		Joda-Time N=2,094		Clo.-Compiler N=17,171	
	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
DiffWMC			-2.330	3.344**			-14.452***	41.577***			-8.302***	-3.841***
			(1.546)	(1.619)			(2.914)	(3.503)			(0.006)	(0.004)
DiffNOC	-37.085***	-93.807***	-58.836***	-152.598***	-0.066***	-0.235***	4.203***	-1.798***	-1.001***	-0.723***	0.617***	2.750***
	(0.359)	(0.372)	(0.037)	(0.043)	(0.020)	(0.007)	(0.113)	(0.040)	(0.007)	(0.021)	(0.0003)	(0.0003)
DiffLCOM	-0.024	-0.008	0.155***	0.179***	-0.016	-0.420	0.217	-1.167	-0.835	-0.255	0.076	0.034
	(0.029)	(0.031)	(0.049)	(0.045)	(0.244)	(0.478)	(1.161)	(1.509)	(0.867)	(2.234)	(0.086)	(0.069)
DiffDIT			-70.763***	-124.104***							0.391 ***	-0.665***
			(0.028)	(0.029)							(0.0003)	(0.0002)
DiffCBO	-2.840	-8.386	1.062	2.261*	-1.162	18.673*	-11.521	-28.867***	5.642***	-7.746***	-7.895***	3.860***
	(5.329)	(5.859)	(0.979)	(1.194)	(3.528)	(10.142)	(12.928)	(9.482)	(0.319)	(0.028)	(0.003)	(0.002)
DiffRFC							8.152	-47.303***				
							(7.483)	(8.690)				
DiffLOC	-0.039	0.039	-0.045	-0.013	-0.009	0.077	-3.004*	5.674***	0.645	1.564	0.615***	0.035
	(0.053)	(0.047)	(0.147)	(0.141)	(0.228)	(0.639)	(1.736)	(1.428)	(0.699)	(2.364)	(0.153)	(0.116)
DiffDelegations	0.001	0.003	-0.005	-0.010***	0.027	0.005	0.201*	0.399***	0.032	-0.074	-0.003	0.002
	(0.003)	(0.003)	(0.003)	(0.003)	(0.043)	(0.066)	(0.104)	(0.100)	(0.051)	(0.117)	(0.002)	(0.002)
DiffSpecInh	-0.371	0.491	0.109	-0.065	-0.489	1.159	-4.686*	0.161	-1.633	-6.170***	0.002	-0.048
	(0.439)	(0.407)	(0.095)	(0.100)	(2.984)	(5.170)	(2.742)	(1.719)	(3.875)	(0.250)	(0.278)	(0.268)
DiffimpInh			0.018	0.341***	-0.461	-0.430	-2.314	-15.482***	-1.313	1.200	0.133	-0.034
			(0.099)	(0.090)	(1.068)	(2.108)	(1.718)	(4.800)	(2.317)	(4.509)	(0.118)	(0.096)
Churns	-0.001	-0.004	-0.001	-0.004**	0.002	-8.872***	-0.015*	-0.026***	-0.006(0.006)	-0.019	-0.001	-0.00001
	(0.002)	(0.004)	(0.001)	(0.002)	(0.006)	(0.001)	(0.008)	(0.007)		(0.013)	(0.0004)	(0.0001)
Constant	-4.183***	-4.082***	-4.048***	-4.043***	-5.312***	-5.346***	-3.345***	-3.323***	-4.545***	-4.462***	-4.624***	-4.654***
	(0.237)	(0.237)	(0.118)	(0.127)	(0.440)	(0.518)	(0.262)	(0.264)	(0.243)	(0.251)	(0.081)	(0.080)

Significance codes: *p<0.1; **p<0.05; ***p<0.01.

variables change the chances of the dependent variable being affected with respect to the reference category—which was set to *"stable"* in our case. As for the columns "↓" of Table 4.3, this means that a negative coefficient for a variable X suggests that for one unit increase of X, the chances that the defect-proneness of source code varies toward a decrease are estimated in the amount indicated by the coefficient, i. e., the higher the coefficient, the higher the chance that the variable contributes to decreasing the defect-proneness of source code. A positive coefficient implies that for one unit increase of X, the chances that the defect-proneness of source code varies toward the stability are estimated in the amount indicated by the coefficient, i. e., the higher the coefficient, the higher the chance of defect-proneness being stable over time. Similarly, in the case of the columns "↑", a negative coefficient for X implies that the chances that the defect-proneness of source code varies toward the stability are estimated in the amount indicated by the coefficient,

i. e., the higher the coefficient, the higher the chance of defect-proneness being stable over time. A positive coefficient would instead indicate that the chances of defect-proneness increasing are estimated in the amount indicated by the coefficient, i. e., the higher the coefficient, the higher the defect-proneness of the source code.

According to this interpretation, the signs of the coefficients for NOC over the various projects did not report a common pattern. For example, in COMMONS-COMPRESS (5th column, 1st row of Table 4.3) we observed a positive coefficient of the variable for "↓" and a negative coefficient for "↑", meaning that the variable statistically influences the stability of defect-proneness over time. On the contrary, on the CLOSURE-COMPILER project (6th column, 2nd row of Table 4.3) the coefficients are positive for both "↓" and "↑", meaning that the variable tends to influence the increase of defect-proneness, overall. As such, we could not delineate a common behaviour for NOC. Likely, its impact depends on the peculiarities of the development process in the different projects rather than on more general aspects.

As for the independent variables considered in our study, namely inheritance and delegation, the discussion is similar. On the one hand, the impact of these metrics is limited to a few projects, suggesting that the defect-proneness of source code is only partially dependent on reusability metrics. On the other hand, the coefficients of the metrics vary without a common pattern. As an example, the coefficient for specification inheritance was positive for "↑" in COMMONS-CLI and negative in JODA-TIME. On the same line, implementation inheritance had a slightly positive coefficient for "↑" in JACKSON-DATABIND, while a negative coefficient in JXPATH. As for the delegation, this turned to be statistically relevant on just two projects, i.e., JACKSON-DATABIND and JXPATH without a consistent sign. Hence, we could conclude that the reusability metrics themselves have a limited connection to defect-proneness. Other indicators, like the structure of the hierarchies computed by NOC, seem to have more statistical power. As such, it is not the amount of reusability mechanisms used by developers to influence the defect-proneness of source code, but rather the way these mechanisms are used in the specific cases. This result has two main implications. First, we could not identify a drawback in the use of inheritance and delegation with respect to software reliability: hence, the application of reusability mechanisms is not per se something to avoid. However, this result represents a call to researchers in software quality, who are required to devise novel quality checkers and/or empirical investigations to monitor the way code reuse is implemented and how it may negatively affect the defect-proneness of source code.

Another valuable consideration can be drawn when considering the control variables. According to our results, none of them seems to be statistically impactful on defect-proneness. We believe this is a relevant result for the software maintenance and evolution research community as a whole. Code quality metrics have been indeed often used to estimate and/or predict defects: our results indicate the lack of statistical significance and possibly imply that the set of metrics considered within defect prediction models should be reconsidered - in this sense, we corroborate previous findings on the limited value of the Chidamber-Kemerer metric suite for defect prediction [141, 161, 287] as well as further stimulate the research on alternative predictors [45, 84, 269, 280].

Key findings of RQ$_2$.

Our findings suggest that the use itself of inheritance and delegation does not influence the defect-proneness of source code. Rather, the specific adoption, e.g., how developers structure the hierarchy of the systems being developed, tends to influence more the likelihood of source code being defective. Furthermore, we found a limited connection between code quality metrics and defect-proneness, possibly revealing that previous research on the relation between metrics and defects should be reconsidered.

RQ*$_3$*. On the impact of reusability mechanisms in code churns

Table 4.4 reports the results obtained when building a *Generalized Linear* model on the data collected for **RQ**$_3$. Differently from **RQ**$_2$, the dependent variable was the code churn, namely a numerical variable.

STATISTICAL MODEL EXPLANATION. The statistical model outputs a single coefficient for each independent variable: this coefficient corresponds to the impact of a one-unit increase on the amount of code churn. Also in this case, the statistically significant coefficients are highlighted with a '*' symbol - a higher amount of '*' implies a higher statistical relevance of a variable with respect to the code churn computed on a defect-fixing commit i. The variables discarded through the multi-collinearity are the same as **RQ**$_2$.

Table 4.4: RQ_3. Results of the statistical model.

	Com.-Codec N=2,134	Com.-Cli N=1,099	Com.-Col. N=3,560	Com.-CSV N=1,634	Comp. N=3,305	Gson N=1,478
DiffWMC	163.951 (105.295)	26.263 (227.457)	-20.375 (56.965)	-20.375 (56.965)	-1,988.919*** (210.722)	-377.039 (269.203)
DiffNOC			10,213.080*** (2,341.143)	-132.074 (1,357.614)	-10,740.970*** (1,699.369)	17,827.570*** (3,288.230)
DiffLCOM	1.383 (2.713)	-12.154 (16.169)	7.799* (4.680)	10.673*** (1.905)	26.285*** (6.021)	-15.488 (12.955)
DiffDIT	3,341.228*** (903.732)	-2,378.489 (1,497.270)	-1,787.167 (1,192.199)		52,852.530*** (2,813.141)	-6,958.231** (2,826.673)
DiffCBO	1,021.357*** (150.161)	-108.063 (134.420)	6,717.225*** (652.191)	-56.282 (94.958)	5,529.115*** (307.003)	1,916.428*** (145.944)
DiffRFC		192.611*** (57.094)	4.682 (60.012)			
DiffLOC	1.293 (2.158)	-9.992* (5.254)	-58.840*** (10.760)	2.994 (2.693)	5.769 (5.471)	46.604*** (10.894)
Delegation	0.017 (0.045)	-0.697*** (0.229)	-0.003 (0.057)	0.165 (0.124)	-0.080*** (0.022)	-0.119** (0.050)
SpecInh	-1.217 (3.145)	39.595*** (11.915)	1.143 (1.211)	-6.889 (6.984)	-0.710 (1.950)	1.415 (1.301)
ImpInh	-0.131 (1.747)	-1.026 (0.870)	-0.386 (4.387)		3.653*** (1.093)	2.080 (2.226)
BugDecrease	1.433 (37.641)	-74.159 (102.978)	-106.139 (620.995)	-1.272 (69.685)	-26.208 (85.330)	-6.128 (90.985)
BugIncrease	-20.191 (37.620)	-70.799 (101.593)	-111.854 (620.996)	-14.892 (69.652)	-15.856 (86.311)	-14.031 (96.408)
Constant	52.948*** (15.918)	126.288** (51.413)	103.349 (77.155)	11.161 (20.003)	91.271*** (25.693)	111.100* (63.054)
	Jack.-Core N=1,543	**Jack.-Datab. N=5,228**	**Jack.-XML N=1,128**	**Com.-JxPath N=598**	**Joda-Time N=2,094**	**Clo.-Compiler N=17,171**
DiffWMC		853.627*** (71.347)		-889.089 (1,208.343)		-35,485.190 *** (1,024.124)
DiffNOC	21,588.520*** (1,212.595)	22,830.430*** (1,564.318)	333.786 (509.649)	24,786.920*** (5,760.288)	54,104.760*** (3,864.815)	204,776.100*** (25,377.820)
DiffLCOM	1.241 (0.765)	-28.862*** (1.208)	-8.269*** (0.992)	21.501*** (5.505)	189.720*** (23.536)	454.243*** (9.841)
DiffDIT		50,782.460*** (1,712.723)				305,846.900*** (27,225.780)
DiffCBO	1,682.687*** (131.899)	3,147.449*** (74.440)	239.229*** (19.892)	3,504.462*** (363.997)	-31,929.670*** (2,814.595)	-472.842 (643.145)
DiffRFC				1,358.532*** (363.735)		
DiffLOC	28.585*** (1.371)	-8.353*** (3.029)	8.568*** (0.946)	-158.255*** (12.875)	344.715*** (42.978)	1,028.922*** (38.580)
Delegation	-0.012 (0.017)	0.010* (0.006)	-0.096** (0.044)	-0.598** (0.294)	-0.580*** (0.107)	-0.014* (0.008)
SpecInh	2.701 (3.677)	-1.281*** (0.305)	-2.847 (2.324)	6.773 (14.133)	-156.445*** (22.026)	0.868 (0.737)
ImpInh		-0.140 (0.499)	3.001** (1.408)	0.942 (3.559)	179.745*** (20.294)	0.406 (0.407)
BugDecrease	83.885* (48.156)	28.236 (19.492)	19.831 (23.738)	-41.147 (149.800)	-625.949 (602.612)	-73.094 (91.940)
BugIncrease	-51.820 (48.181)	-7.043 (20.359)	-20.624 (26.005)	14.086 (149.626)	-690.321 (618.444)	-52.403 (91.954)
Constant	25.537 (47.505)	182.664*** (57.987)	57.931*** (9.413)	1,561.449*** (387.616)	-6,729.363*** (706.442)	89.824 (76.824)

Significance codes: *p<0.1; **p<0.05; ***p<0.01.

STATISTICAL MODEL ANALYSIS. Looking at the Table 4.4, we can draw various conclusions. As expected, the LOC metric was found to be statistically significant in 9 systems out of 12. The coefficients are also relatively high in all cases, meaning that larger classes are typically harder to maintain - in this respect, we could corroborate previous findings in literature [140, 318]. The CBO metric, which computes the coupling between objects, was also statistically significant in nine projects, confirming that developers spend more effort in fixing defects pertaining to highly-coupled classes [188]. Other code quality metrics were not statistically significant. So, in conclusion of this first point of discussion, we could report that, besides LOC and CBO, the role of code metrics to estimate the maintenance effort seems to be limited. Once again, this finding is of the interest of the software maintenance and evolution research community, which might be called to define novel metrics and/or instruments to monitor maintenance effort over time.

Turning the focus on our independent variables, we could find similar conclusions as in **RQ**$_2$ when considering inheritance. Both specification and implementation inheritance were indeed most not statistically significant, with some exceptions. The former was relevant for the projects COMMONS-CLI, JACKSON-DABIND, and JODA-TIME. However, the sign of the coefficients revealed that the metric was statistically related to the increase of code churn only in the case of COMMONS-CLI. By analyzing this case further and relating the statistical result with the trend analysis conducted in **RQ**$_1$, we could better understand the reason behind this correlation. Most of the defects available for COMMONS-CLI were introduced and fixed after the design erosion discussed in **RQ**$_1$. It is therefore reasonable to believe that it was the lack or the decrease in the use of inheritance mechanisms which caused a higher maintenance effort when fixing defects. This interpretation is in line with what observed on the other systems, i.e., JACKSON-DABIND and JODA-TIME, where the specification inheritance was negatively correlated to maintenance effort, meaning that this was a significant factor to reduce the code churn required to fix defects.

Implementation inheritance was found to be statistically relevant in just two cases, i.e., on JACKSON-DATABIND and JXPATH. While in the former case the coefficient was close to zero—indicating little to no correlation to the dependent variable—, it was of -15.482 in the second case. Hence, also in this case we could conclude that this metric was negatively correlated to the maintenance effort. Enlarging the discussion to the other inheritance metrics subject of the study, namely NOC and DIT, we could discover similar results as **RQ**$_2$. Both NOC and DIT were positively correlated to the dependent variable and the coefficients were relatively large in all cases: these results imply that the structure of hierarchies might strongly influence the maintenance effort to fix defects, hence corroborating the results obtained in our previous research question, other than the results of empirical studies reporting how NOC and DIT could worsen software maintainability [77, 78, 282].

As for delegation, the coefficients were mostly negative, even if relatively small. Hence, we could conclude that there exist a small negative correlation between the metric and maintenance effort, which implies that the use of delegation may decrease the overall amount of code churn required to fix defects.

Key findings of RQ$_3$.

Reusability metrics mostly reduce the effort required to fix defects, as measure by code churn. Also in this case, we found that the structure of the hierarchies might affect more maintenance effort than the mere use of inheritance. Finally, the lines of code and coupling between classes represent factors that strongly influence the maintenance effort.

4.4 THREATS TO VALIDITY

A number of potential threats might have biased the study. This section discusses them and reports the mitigation strategies applied.

THREATS TO CONSTRUCT VALIDITY. Threats in this category refer to a possible mismatch between theory and observation. In this respect, the selection of the dataset represents a crucial point for which there are various observations and remarks to make. We used DEFECTS4J (version 2.0.0), which has been already widely used by the research community in several previous studies (e.g., [157, 277, 321]) and that reduced possible bias due to the presence of uncontrolled conditions, e. g., tangled changes [148], allowing us to investigate the impact of reuse mechanisms on defect-proneness and maintenance effort more precisely.

As for the defects considered, the GIT repositories of the considered projects may contain more issues than those reported in DEFECTS4J. Nevertheless, there are two observations to make in this respect. i), a notable amount of these issues do not actually pertain to defects but to other maintenance and evolution tasks. For instance, let us consider the case of the COMMONS-COLLECTIONS project, i.e., the project having the least amount of defects in our study. According to the issue tracker,[5] the project has a total of 787 issues (filtering by Type='All' and Status='All'): of those, only 374 pertain to defects (filtering by Type='Bug' and Status='All'), while the remaining 413 issues refer to enhancements, implementation of new features, and other evolutionary tasks. As such, the set of candidate defects that we might have considered is much lower in size with respect to the raw data reported on the issue trackers. ii) A number of issues do not report reliable information. Still taking the COMMONS-COLLECTIONS project as an example, we noticed that 159 of the issues marked as 'Closed' or 'Resolved' (filtering by Type='Bug' and Status='Resolved, Closed') report the strings *"Invalid"*, *"Not a Bug"*, *"Won't Fix"*, *"Cannot Reproduce"*, and *"Duplicate"* as actual resolution, hence indicating that these defects were false positives, not taken into account by the developers, or already addressed as part of duplicated issue reports. As a conclusion, we found out that issue trackers contain a non-

[5] The COMMONS-COLLECTIONS issue tracker: `https://issues.apache.org/jira/projects/COLLECTIONS/issues/`.

negligible amount of noise that would require substantial filtering and data quality procedures, which is indeed what DEFECTS4J guarantees.

Still reasoning on the number of issues reported on the issue trackers of the considered systems, it is worth remarking that the candidate set of defects was limited by the types of defects and the types of fixes performed. We should distinguish multiple cases. First, some defects may not pertain to production code, e.g., test code defects, or might relate to the update of third-party libraries or configuration files. As explained in Section 4.2, these defects were not considered by DEFECTS4J and, as a consequence, by our work. However, these defects would have not created any noise for our analysis: indeed, our work aims at understanding how reusability metrics affect the defect proneness of the production code and, for this reason, all the defects that are not related to production code cannot affect our measurements. Second, some defects might not be verifiable or not traceable, even though they relate to the production code. As for the former, they might either represent true defects that developers did not have enough time to deal with or false positives, namely defects that developers ignored and that were marked as 'Resolved' or left opened in the issue tracker without any further action: considering these defects in our analysis would have caused some degree of uncertainty in terms of number of defects considered and, for this reason, we would have likely introduced some bias. As for the latter, these are defects that we could not trace back in the history of the considered projects and, as such, we could not technically analyze without approximation or heuristics that would have, again, introduced some degree of uncertainty. Last but not least, the candidate set of defects might have been limited by the types of fixing activities: DEFECTS4J indeed discards defects whose fixes were performed along with other maintenance and evolution activities, e.g., tangled changes. Among the various cases discussed, this latter was the most critical in our case, as it refers to real defects that were not considered in the scope of the analysis and that might have biased the computation of the number of defects in the change history of the projects considered. A systematic assessment of the noise caused by these missing defects would have required the definition of dedicated data quality protocols through which we could have (I) systematically classified real defects among those not considered by DEFECTS4J; (II) analyzed the corresponding fixes to understand their nature; and (III) assessed the extent to which our findings varied when considering the newly classified defects. To the best of our knowledge, the current literature does not offer any (semi-)automated instrument to perform a similar assessment nor guidelines to follow. We deemed the research investigation and methods required to perform such a systematic assessment as out of scope. Nonetheless, to partially analyze the potential noise given by those missing defects, we have attempted to estimate the noise of our analysis in the case of the COMMONS-COLLECTIONS project through a simple, likely suboptimal approach based on text mining and manual analysis. We first (i) mined the summary of each of the 215 marked as 'Closed' or 'Resolved' defects having as resolution the string *"Fixed"*, and (ii) used a keyword-based approach to classify those issues according to their type. More specifically, we classified an issue as 'test-related' if the summary contained the keyword *"test"*, as 'documentation-related' if it contained keywords such as *"JavaDoc"* and *"comment"*, and as 'configuration-related' if it contained keywords such as *"JDK"*, *"compil*"*, *"build"*, and *"CI"*. In this way, we could estimate the amount of issues whose fixes did not modify the production code, hence covering the first case described above. Afterwards, we manually went through the summaries of the remaining issues to assess how many of them revolved around modifications that were not verifiable, not traceable, or that performed modifications other than defect fixes—hence covering the other possible cases of noise. As a result, we discovered that 181 issues were not considered within DEFECTS4J. Among them, 1% referred to Continuous Integration concerns, 7% to JDK compilation issues, 13% to test code defects, e.g., flaky tests, and 17% to documentation issues, e.g., unclear JavaDoc comments. Hence, 69 of them (38%) of the discarded defects did not concern production code. From the subsequent manual analysis, we discovered that 21 were untraceable

(19%), while 84 were issues raised by specific users that the maintainers of the system solved by recommending configuration changes, hence not making any change to the system itself (46%). The remaining 7 defects were not correctly classified by the keyword-based approach and pertaining to documentation or configuration issues - in these cases, the summaries reported keywords different from those used by the classifier, e.g., *"typo"*. Perhaps more interestingly, we found that 34 defects matched the requirements of DEFECTS4J: yet, six were reported between November 2020 and June 2023, namely after the release of DEFECTS4J 2.0.0 (issued on September 15, 2020), while 24 were part of the defects deprecated by DEFECTS4J. As such, the set of defects actually analyzable was four, which is exactly the number of defects we analyzed. While such an additional analysis was not performed on all the considered systems, it let us provide some insights on the noise possibly affecting our results. While we acknowledge that our study took into account only a subset of defects having specific properties, it actually contains most of the real defects that should be taken into account. The noise caused by the presence of additional issues on the issue trackers is likely to be limited, as most defects and corresponding fixes are not related to production code. In conclusion, we argue that our conservative approach in terms of defect selection, i.e., that of relying on the defects pointed out by DEFECTS4J, represents the best option to properly measure the extent to which reusability mechanisms impact the defect proneness of source code. As a side result of our additional analysis, we could also further corroborate the validity of DEFECTS4J - which we consider as a valuable outcome for our research community.

A second threat to validity relates to the selection of the metric used to operationalize maintenance effort. We used code churn [251]: we are aware that this metric can only proxy the actual effort spent when maintaining source code, yet this choice is required in our case because of the unavailability of precise data regarding the maintenance effort in our dataset. Nonetheless, proxy measurements are still used and considered in the field[313]. The tool we used to extract metrics, e. g., reusability or CK metrics, represents another potential threat to validity. We used tools already validated and used by the research community [118, 327]. Finally, as mentioned in Section 4.2, in DEFECTS4J a single bug can be introduced by multiple factors, but its resolution will always occur within a JAVA file. Thus, to avoid possible threats to contraction validity, we discard commits that introduced defects caused by issues not involving source code. This allowed us to only focus on defects introduced and resolved through changes to the source files.

THREATS TO INTERNAL VALIDITY. These threats refer to factors that might have impacted the study results. In our context, these might be connected to the selection of the metrics used to build the statistical models. On the one hand, we were interested in understanding the role of reusability metrics and, for this reason, we operationalized implementation and specification inheritance, other than delegation, following their exact definition. On the other hand, we used control variables previously shown to be significantly correlated to source code quality [67, 78, 332, 334]. Through these actions, we could rely on a set of independent variables and control metrics that come from either our working hypotheses or the state of the art.

THREATS TO CONCLUSION VALIDITY. Threats related to this category refer to the selection and the use of the statistical test. When addressing **RQ**$_2$ we modeled the problem using a *Multinomial Logistic Linear* model [340], while we built a *Generalized Linear* model [95] in the context of **RQ**$_3$. These choices come from the nature of our response variables, i. e., multiclass and continuous, respectively. Moreover, the research community used these types of model in similar contexts [57, 118, 182]. The empirical analysis conducted in this study had a quantitative connotation and, in particular, we sought to understand the relation between code reusability and defects through statistical modeling. Nonetheless, we are aware that more qualitative investigations aiming at linking the root cause of defects with the reuse mechanisms might potentially reveal further insights into the matter. While a more complete overview of this type is part of our future research agenda, in the context of this work we already provided some preliminary insights through the manual analysis discussed. Such an analysis was in line with the statistical conclusions drawn when addressing **RQ**$_2$ and **RQ**$_3$.

THREATS TO EXTERNAL VALIDITY. As for the generalizability of the results, the main threat might be connected to the target of our work. In particular, we focused on 12 JAVA projects having more than 44,900 commits and coming from the DEFECTS4J dataset. As such, our work was based on the analyses conducted on a *sample*: our generalization strategy can be identified within the *sample-based generalization* strategies proposed by Wieringa and Daneva [373]. In particular, among those strategies, the "statistical learning" seems to be the most appropriate. Wieringa and Daneva [373] reported that the *"descriptions of statistical sample phenomena can be used to predict similar phenomena in new samples. [...]. The goal is not to generalize to a population, but to generalize to the next few cases"*. This strategy is basically in line with the *generalizing by similarity* principle described by Ghaisas *et al.* [114]. When contextualizing those strategies in our case, it is likely that similar results might be obtained in projects having similar characteristics with respect to those analyzed in our work (see Table 4.1). Therefore, we cannot claim the generalizability of our findings to projects having different properties or even written in different programming languages. Replications in these contexts would still be desirable and already part of our future research agenda.

5 THE YIN AND YANG OF SOFTWARE QUALITY: ON THE RELATIONSHIP BETWEEN DESIGN PATTERNS AND CODE SMELLS

5.1 INTRODUCTION

The idea of design patterns was proposed in 1995 by the *Gang of Four*, who defined them as reusable solutions to commonly occurring problems that arise during the design and development of software applications [107]. Adopting such reusability mechanisms can provide several advantages from the developers' perspectives, as their flexibility makes them re-appliable by changing the context, the environment, and the programming languages, without changing the philosophy driving a given pattern [361]. The large spread of Object Oriented programming languages boosted developers to reuse instance classes and to create hierarchies that can be easily used as a basis for the introduction of design patterns. Previous studies investigated the use of design patterns in JAVA [46, 133, 312], mainly because (I) JAVA offers, by design, mechanisms and data structures that make large use of reusability principles, especially linked to inheritance, and (II) although the fluctuating trends, JAVA is still one the most adopted programming languages in large companies and open-source communities.[1] While most research emphasize the importance of reusability mechanisms to guarantee high quality of the software, a number of studies seem to go in the opposite direction, highlighting that a sub-optimal implementation of design patterns can, in turn, increase the code complexity and negatively impact the code in terms of maintainability and comprehension [173]. Fowler and Beck identified *code smells* as

[1]Source: `https://www.tiobe.com/tiobe-index/`

indicators of the poor quality of code, affecting its cohesion, coupling, and comprehensibility, ultimately making the code difficult to maintain [105].

In this chapter, we investigate the role that design patterns play in the presence of code smells.

We analyzed 15 open-source JAVA projects spanning over 542 releases, by extracting information about the instances of design patterns and the code smells affecting the classes, and assessing (I) the co-occurrences of instances of design patterns and code smells, and (II) whether the presence of design patterns instances is correlated with the formation of code smells.

We find that, although design patterns are intended to improve the quality of the code, as they represent a reuse mechanism, there is no guarantee on them enhancing the goodness of the software; on the contrary, design patterns can in fact determine the appearance of code smells in certain cases. We point out the importance of carefully dealing with design patterns by applying them properly and monitoring their evolution in the software projects.

Our main points of contribution can be summarized as follows:

1. An empirical investigation of design patterns and their impact on code smells, that can enhance the state of the art on software reusability and, at the same time, can aid practitioners in monitoring the changes in complexity and comprehension when implementing design patterns;

2. A publicly available online appendix containing all the scripts, raw data, and additional materials used to perform our experiments, that can be leveraged for replication and verification of our work [412].

5.2 RESEARCH QUESTIONS AND METHOD

The *goal* of this study was to understand whether and how design patterns are related with code smells. The *context* consisted in 10 design patterns, *i.e.*, *Adapter/Command*, *Bridge*, *Singleton*, *Template Method*, *Proxy*, *State/Strategy*, *Decorator*, *Factory Method*, *Component*, and *Observer*. The

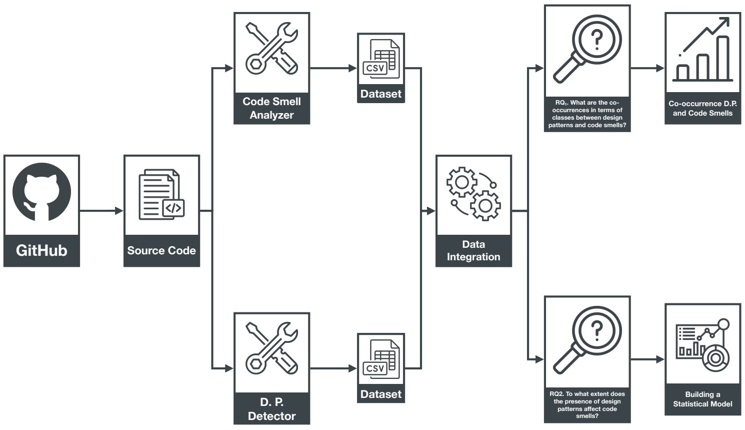
GitHub
Source Code
Code Smell Analyzer
CSV
Dataset
D. P. Detector
CSV
Dataset
Data Integration
RQ$_1$. What are the co-occurrences in terms of classes between design patterns and code smells?
Co-occurrence D.P. and Code Smells
RQ2. To what extent does the presence of design patterns affect code smells?
Building a Statistical Model

Figure 5.1: Overview of the research method applied in this work.

perspective was of both researchers and practitioners, as the former are interested in increasing the body of knowledge on this topic, and the latter are concerned about understanding how design patterns impact code quality in software systems.

Based on our *goal*, we formulated two research questions.

> **RQ$_1$.** *What are the co-occurrences in terms of classes between design patterns and code smells?*

RQ$_1$ aimed at comprehending the fluctuations in the frequency of classes participating in a particular design pattern, and the co-occurrence of code smells in such classes. We wanted to assess whether classes implementing design patterns contain code smells themselves, and we expected to see a low frequency of smells in classes participating in design patterns.

We were interested in understanding whether and how design patterns are correlated with the presence of code smells.

For this reason, we asked:

> **RQ$_2$.** *To what extent does the presence of design patterns affect code smells?*

To answer our research questions, we performed an empirical study (I) analyzing the co-occurrences of code smells in classes participating in design patterns, and (II) applying statistical models to understand the impact of design patterns on the emerging of code smells.

Figure 6.1 depicts the method applied in this work, which we designed following the guidelines by Wohlin et al. [376] and the *ACM/SIGSOFT Empirical Standards*;[2] in particular, we leveraged the "General Standard", "Data Science", and "Repository Mining" guidelines. We selected 15 JAVA projects from GITHUB and manually built 542 releases. Then, we extracted design patterns instances in the projects by leveraging the detection tool proposed by Tsantalis *et al.* [346], and we identified code smells affecting the projects by running DECOR [244] on each release. We combined these pieces of information to understand, on the one hand, the co-occurrences of classes that collaborate into design patterns and, simultaneously, are involved in some code smell. On the other hand, we investigated whether design patterns affect code smells from a statistical standpoint.

In the following, we report the detailed design of our work. The complete dataset, scripts, and raw results of our study are available in the online appendix of this thesis [412].

Dataset Collection

Table 5.1: Overview of the projects analyzed.

Project Name	Description	Stars	Forks	N. Releases	N. Releases Analyzed	LOC	Link
Arthas	Java Diagnostic Tool	31,9k	6,9k	47	43	61K – 46K	https://github.com/alibaba/arthas
Apollo	Configuration Management System for Microservices	27,8k	10,1k	38	8	88K – 90K	https://github.com/apolloconfig/apollo
Caffeine	High Performance Caching Library	13,2k	1,4k	65	9	70K – 83K	https://github.com/ben-manes/caffeine
Data Transfer Project	Transfer Data Online	3,4k	442	55	47	40K – 41K	https://github.com/google/data-transfer-project
ApkTool	Reverse Engineering	15,8k	3,3K	16	14	15K – 18K	https://github.com/iBotPeaches/Apktool
JSQL Parser	RDBMS agnostic SQL	4,2k	1,2K	30	4	50K – 55k	https://github.com/JSQLParser/JSqlParser
Disruptor	High Performance Inter-Thread Messaging Library	15,8k	3,8K	13	6	20K – 20K	https://github.com/LMAX-Exchange/disruptor
Mockito	Framework for Unit Tests	13,7k	2,4K	198	112	89K – 88K	https://github.com/mockito/mockito
MyBatis-3	SQL mapper framework for Java	18,2k	12,1K	39	13	100K – 98K	https://github.com/mybatis/mybatis-3
Eureka	AWS Service Registry	11,7k	3,7K	146	109	50K – 53K	https://github.com/Netflix/eureka
Hystrix	Latency and Fault Tolerance Library	23,2k	4,7K	79	40	75K – 48K	https://github.com/Netflix/Hystrix
Zuul	Gateway Service	12,5k	2,3K	5	4	35K – 31K	https://github.com/Netflix/zuul
RxJava	Library for Composing Asynchronous and Event-Based Programs for Java-VM	46,8k	7,7K	231	101	41K – 42K	https://github.com/ReactiveX/RxJava
Jadx	Dex to Java Decompiler	33,6k	4,2K	27	19	118K – 70K	https://github.com/skylot/jadx
Spring Data JPA	Data Access Layer Simplify	2,6k	1,2K	78	13	45K – 44K	https://github.com/spring-projects/spring-data-jpa

Table 5.1 provides an overview of the dataset used in this work. We defined two main criteria for the selection of the projects to consider:

[2]Available at: https://github.com/acmsigsoft/EmpiricalStandards

BUILD AVAILABILITY. We selected only JAVA projects that can be built without errors. This criterion is driven by the constraints dictated by the design pattern detection tool by Tsantalis *et al.* [346], which we selected to obtain data on the design patterns implemented in the projects. The tool requires that target projects build without errors, as it leverages JAVA BYTECODE to generate an intermediate code representation. Thus, it can only be executed on projects that can be built successfully. To ensure that, we manually compiled the candidate projects and assessed their compliance with this criterion.

NUMBER OF STARS ON GITHUB. We selected projects with a minimum number of stars of 2K on GitHub. We set such a threshold to avoid the inclusion of toy projects or personal projects developed by users. The number of stars has been demonstrated to be a good proxy metric to estimate the popularity of repositories and their overall quality [291].

Considering the points above, we manually identified GitHub projects meeting the criteria. Due to the time-consuming activity, we limited our search to the first 10 pages of GITHUB results filtered to JAVA, finding 45 candidate projects. Starting from the initial set of candidates, the first and second authors manually set up the projects leveraging the build system and directions provided in the corresponding GITHUB repository. However, 50% of the projects could not be successfully configured and built, due to incompatibility problems with the versions of some libraries. This issue is not uncommon in the context of mining software repositories, and was pointed out by Hassan *et al.* [138] when they performed a comparison among the main JAVA building systems. After filtering out the projects which could not be built, we were left with 23 candidates. To avoid considering projects irrelevant to our research questions, we ran the design pattern detection tool [346] and discarded projects containing no instances of design patterns. At the end of this process, we had identified 15 JAVA meeting the selection criteria and useful to our experiments, reported in Table 5.1.

Design Pattern Extraction

To extract the design pattern instances implemented in the considered projects, we leveraged the tool by Tsantalis *et al.* [346], which we selected on the basis of two main aspects:

DETECTION CONFIDENCE. The tool can detect 10 kinds of design patterns, i. e., *Adapter/Command*, *Bridge*, *Singleton*, *Template Method*, *Proxy*, *State/Strategy*, *Decorator*, *Factory Method*, *Component*, and *Observer*, with 100% precision and no false positives [346]. These performances make the tool state-of-the-art for the task of design pattern instance detection. However, due to the identical UML structure of *Adapter/Command* and *State/Strategy*, the tool aggregates them into a single type, as they cannot be distinguished by an automated process [346].

FLEXIBILITY TAKEN INTO ACCOUNT. Due to the internal implementation of the tool, it can also identify custom implementation of known design patterns types.

To perform its task, the tool executes a number of steps. First, it analyzes the characteristics of the projects in terms of associations, generalizations, method invocations, and so on. At the end of this step, an *n x n* adjacency matrix is generated, where *n* represents the number of classes. The tool identifies the inheritance hierarchies among the classes, considering all kinds of inheritance implemented in JAVA, i. e., specification inheritance, implementation inheritance, and abstract classes, and leverages them to build a tree modelling the hierarchical structure of the project. Such a tree generates one or more subsystems, which are then provided as input to a similarity score algorithm. The algorithm compares the identified subsystems with the structure of design patterns.

Code Smell Detection

To detect the code smells affecting the considered projects, we used DECOR [244], as previously done in similar work [81, 136, 156] because it represents a compromise between execution time and performance [34, 244, 274], reporting 100% recall, and precision greater than 50%.

In particular, DECOR employs a combination of multiple heuristic approaches to detect code smells in source code. Given a class A, the tool considers A to be affected by a code smell S *if and only if* for each metric used to estimate the presence of S, the following condition is verified: $metric_i \geq threshold_i$. The higher the difference between $metric_i$ and $threshold_i$, the greater the intensity of *S*.

We leveraged DECOR to detect three smells, i. e., *Complex Class*, *God Class*, and *Spaghetti Code*, as they represent the quality of the code in terms of understandability.

***RQ*$_1$*: Analyzing the Co-Occurrences of Design Patterns and Code Smells*

To answer **RQ**$_1$, we calculated the frequency of classes participating in design patterns and, simultaneously, being affected by code smells. We merged the data coming from the execution of the design pattern detection tool by Tsantalis *et al.* [346] and the code smell detector DECOR. We computed the number of classes that are involved in some design pattern and, at the same time, are affected by code smells. We normalized all the results using MIN-MAX in the range $[0;1]$, and plotted the frequency of co-occurrences by means of heatmaps.

***RQ*$_2$*: Correlation between Design Patterns and Code Smells*

To address **RQ**$_2$, we built a statistical model analyzing the impact that the presence of a design pattern has on the emerging of code smells. In the following, we report the independent, dependent, and control variables involved in the analysis with the statistical model.

INDEPENDENT VARIABLES. We were interested in understanding whether and to what extent the presence of design patterns impacts code smells. For this reason, we considered design patterns as independent variables. We focused on 10 design patterns, i. e., *Adapter/Command, Bridge, Singleton, Template Method, Proxy, State/Strategy, Decorator, Factory Method, Component,* and *Observer*. The selection of such set of design patterns was driven by their availability, as they can be extracted by the design pattern detection tool by Tsantalis *et al.* [346]. To avoid possible threats to validity, we considered the same aggregation on design patterns made by the authors of the tool; as the patterns *Adapter/Command* and *State/Strategy* share the same UML structure, it is not possible to automatically distinguish them by means of a tool.

DEPENDENT VARIABLES. As we aimed to understand the impact of design patterns on the emergence of code smells, the presence of code smells affecting the code represented the dependent variable in our study. We focused on three code smells [105], i. e., (I) *God Class*, affecting a class that implements several responsibilities, and it is invoked by most of the system to perform their actions, (II) *Spaghetti Code*, representing a class that implements long methods without parameters, and (III) *Complex Class*, that is a class being hard to understand and showing a high level of cyclomatic complexity. The principal reason driving the selection of such smells is given by the claims made in previous studies about them being representative of code complexity and comprehensibility, which are perceived as crucial for maintenance tasks in the perspective of developers [8, 9, 173, 264]. Likewise, we decided not to consider additional known code smells, such as *Parallel Inheritance, Middle Man*, or *Refused Bequest* due to issues in the detection mechanisms. They have been formerly identified leveraging a custom version of DECOR that implements a dynamic approach for the detection [202]. Unfortunately, this version is not publicly available.

CONTROL VARIABLES. Conscious that external unconsidered factors can impact the fluctuation of the dependent variable, we considered a set of code quality metrics as control variables for our experiment, to avoid possible threats to the conclusion validity of our study. We selected five control metrics, i. e., *Lines of Code* (LOC), *Lack of Cohesion of Methods* (LCOM), *Number Of Attributes* (NOA), *Weighted Methods per Class*

(WMC), and *McCabe's Cyclomatic Complexity* (CC); these metrics have been demonstrated to be good estimators for code quality [332, 334]. We extracted the control metrics by using DECOR; however, we remark that DECOR does not consider such variables during the estimation of the presence of code smells, which means that there is no direct correlation between the dependent and control variables of our study [14].

We manually assessed the possibility of multi-collinearity among the variables involved in our study, to avoid threats to the validity of our work, as explained in the following.

STATISTICAL MODEL. Given the nature of the dependent variable, i. e., the presence or the absence of a certain code smell, the *Generalized-Linear-Model* was used [339]. We selected this statistical model because it can be applied to estimate nominal variables that can assume two levels. We built the model using the *multinom* function provided by the *nnet*[3] package in R. Before running the statistical model, we took into account the multi-collinearity problem, occurring when two or more independent variables are bounded with a high level of correlation, and one of them can be used to predict the other. The presence of multi-collinearity among variables can bias the results, therefore, we followed the guidelines proposed by Allison *et al.* [12] to mitigate this threat. We did not remove any independent variable, because the standard error was, in any case, lower than 0.9, and interpretability problems arise with a standard error higher than 2.5 considering 95% prediction interval [232].

5.3 ANALYSIS OF THE RESULTS

In this section we report the results of our study and discuss about the implications of our findings.

[3] `https://cran.r-project.org/web/packages/nnet/nnet.pdf`

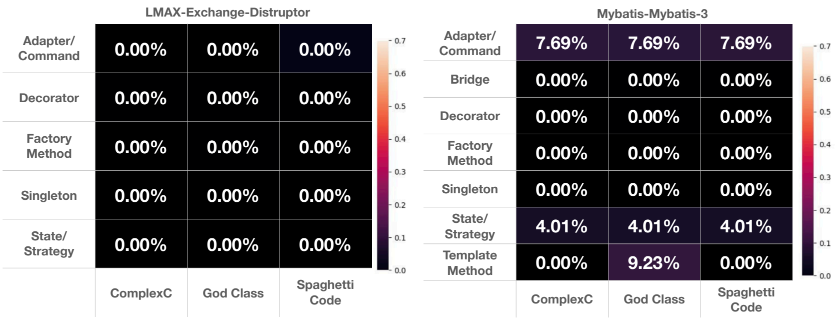

Figure 5.2: Co-occurrences of code smells and design patterns.

RQ1. On the Co-occurrence of Design Patterns and Code Smells

RQ$_1$ focused on comprehending whether and to what extent design patterns and code smells co-occur in the same classes.

Figure 5.2 provides an overview of the results obtained from the co-occurrences analysis. We report the data related to four projects, and we make the complete results available in the online appendix [412]. The figure depicts a heatmap reporting the extent to which classes participating in design patterns contained code smells. For example, in the MYBATIS-3 project, 7.69% of the classes participating in an instance of *Adapter/Command* were affected by the *God Class* smell.

The results obtained from the 15 analyzed projects were variegated, hinting at the observation that the co-occurrence of design patterns and code smells may vary depending on the project. By analyzing the frequencies reported in each heatmap, we noticed that two main patterns emerged, describing two families of projects. The first kind of project was characterized by design pattern instances completely free from code smells. That was the case of projects APOLLO, APKTOOL, DATA TRANSFER PROJECT, JSQL PARSER, DISRUPTOR, MOCKITO, and SPRING DATA JPA. In these projects, none of the classes participating in design patterns were affected by code smells. The opposite pattern arose from a set of seven projects which presented a high frequency of co-occurrence of design patterns and code smells, *i.e.*, HYSTRIX, CAFFEINE, MYBATIS-3, EUREKA, RXJAVA, JADX and

ZUUL. Such projects exhibited code smells affecting classes participating in design patterns, and in each project the threatened types of design patterns went from two to four. A single project, *i.e.*, ARTHAS, presented only one co-occurrence, in fact the *State/Strategy* pattern was the sole affected by the *God Class* and *Spaghetti Code* smells.

An interesting observation emerged from the analysis of co-occurrence, which showed that the *State/Strategy* pattern was touched by code smells in every project—except those not presenting any co-occurrence. In particular, in all the projects revealing at least or exactly one co-occurrence, classes implementing the *State/Strategy* pattern were affected by the *God Class* smell, in eight projects they also showed *Spaghetti Code* issues, and in four projects they presented *Complex Class* smells. We conjecture that this result is driven by the characteristics of the *State/Strategy* pattern itself, as its goal is to provide different behaviors depending on the current state of an object [107]. We suppose that as the behaviors to implement grow in number and size, the complexity of the involved classes also tends to increase. This observation remarks the non-triviality of the use of design patterns to enhance code quality and maintainability; as design patterns themselves risk being affected by the problems they aim at avoiding. The *Adapter/Command* pattern showed a similar trend to the *State/Strategy* one, as it was threatened by *God Class* and *Spaghetti Code* in five of nine projects, and by *Complex Class* in two projects. We observed that instances of the *Singleton* and *Bridge* patterns appeared in co-occurrence with code smells in four projects, followed by the *Template Method*, which emerged in three projects. Classes implementing the *Observer* and *Decorator* patterns were affected by code smells in two projects, while *Factory Method* implementations resulted being smelly in one project.

Key findings of RQ$_1$.

Classes participating in design patterns may be affected by code smells, resulting impacted by the same problems they are supposed to avoid. The *State/Strategy* pattern emerged in all projects as being threatened by code smells, followed by the *Adapter/Command* pattern, which resulted being compromised in six projects.

RQ2. On the Impact of Design Patterns on Code Smells

With our second research question, we aimed at assessing how the presence of design patterns impacts the arising of code smells. By performing statistical analysis, we found that most design patterns did not influence the code into being affected by code smells. However, in nine cases, the analysis revealed that the implementation of design patterns determined the presence of code smells in a statistically significant way. Table 5.2 reports the complete results of the statistical analysis. The first observation we noticed studying the results was related to the *State/Strategy* pattern, as it appeared as the most co-occurring with code smells in the first phase of our research. Nevertheless, the statistical analysis revealed that its presence did not significantly affect the emerging of *Complex Class* and *God Class* smells, but only determined the code being *Spaghetti*. On the other hand, the *Adapter/Command* pattern turned out to be significant in the occurrence of the *God Class* smell and in a minor manner also for the *Spaghetti code*, in concordance with the results observed in RQ$_1$.

The presence of *God Class* was significantly affected also by the *Bridge*, *Singleton*, and *Template Method* patterns, although the co-occurrences were found in a few projects.

In contrast with the purposes of design patterns, which include guaranteeing code maintainability and comprehension, we found that their presence often leads to the introduction of code smells, which are signs of poor implementation practices instead. This led us to reflect on the importance of properly designing and applying best practices for code maintainability, as the effects of our choices can produce unexpected outcomes. We observed that, although design patterns are supposed to make the perfect code, they can be determinant for the arise of code smells. We conjecture that the motivation behind this phenomenon can be connected with the intention driving the introduction of design patterns; attempting to reorganize the code to make it better structured, developers actually introduce degrees of complexity, ultimately leading to code smells threatening the overall program comprehension.

Key findings of RQ$_2$.

The presence of design patterns does not regardless guarantee good quality, as they can be affected by code smells. In particular, the presence of a *God Class* can be associated with a number of patterns, such as *Adapter/Command*, *Bridge*, *Singleton*, and *Template Method*.

5.4 THREATS TO VALIDITY

In this section, we recognize the possible threats that could impact the results of our study, and discuss the mitigation strategies that we applied.

CONSTRUCT VALIDITY. Construct validity refers to the relationship between *theory* and *observation*. The main concern regards the selection of the dataset leveraged in the experiments, as the choice of the dataset can influence the observed results. To mitigate this aspect, we adopted a rigorous process to select projects based on empirical evidence of their characteristics. On the one hand, we selected only popular projects publicly hosted on GITHUB, estimating their popularity based on the number of stars. On the other hand, we only selected projects for which a building system was provided, and we manually inspected projects to ensure compatibility with the tools adopted to extract design patterns and code smells. Another possible threat to construct validity is concerned with the tool leveraged to extract data on dependent, independent, and control variables. To mitigate this aspect, we chose the state-of-the-art

Table 5.2: Results of the statistical model concerning the relationship between design patterns and code smells.

Design Pattern	Complex Class	God Class	Spaghetti Code
Adapter/Command	8.341	0.594^{***}	-0.384^{*}
Bridge	17.129	17.129^{***}	0.265
Component	166.346	1.544^{***}	-12.602
Decorator	125.670	-0.189	-0.808
Factory Method	1,063.142	-11.255	1.157^{**}
Observer	68.508	1.801	1.380
Proxy	1,008.371	-11.891	-10.424
Singleton	93.883	1.722^{***}	-2.392^{**}
State/Strategy	-4.897	-0.036	0.349^{***}
Template Method	-16.859	0.586^{***}	-0.146
LCOM	-0.0001	-0.0003^{***}	-0.0003^{***}
LOC	0.014	0.007^{***}	0.007^{***}
McCabe	7.598	-0.005^{***}	0.002^{***}
NOA	-0.088	0.034^{***}	0.004^{***}
WMC	-0.008	0.074^{***}	0.033^{***}

Significance: $^{*}p<0.1$; $^{**}p<0.05$; $^{***}p<0.01$.

tools (I) to extract code smells and CK metrics, *i.e.*, DECOR, and (II) to detect design patterns, *i.e.*, the tool proposed and validated by Tsantalis *et al.* [346]. Although it comes with possible imprecision, i. e., design patterns sharing the same UML structure (*Adapter/Command* and *State/Strategy*) are considered the same, it still represents the state-of-the-art for design pattern detection.

INTERNAL VALIDITY. Threats to internal validity are factors that could influence the observed results. In order to avoid threats affecting the statistical model employed to answer RQ_2, we kept an eye on CK metrics, which acted as control variables.

CONCLUSION VALIDITY. The major threat to conclusion validity regards the application of statistical models to answer our second research question. We selected the Multinomial Logistic Linear model [339] due to the nature of the problem, and we also considered possible multi-collinearity to avoid any interpretation bias.

EXTERNAL VALIDITY. Threats to external validity are linked to the generalizability of the observed results. We analyzed 542 releases of 15 different projects in terms of scope and size. We are aware that generalizability can depend on multiple aspects, such as programming language; however, as part of our future work, we plan to extend this study, considering a broader set of projects to analyze, selecting them according to a variety of programming language, domain, and size.

6 UNDERSTANDING DEVELOPER PRACTICES AND CODE SMELLS DIFFUSION IN AI-ENABLED SOFTWARE

6.1 INTRODUCTION

As previously in previous Chapters, the presence of code smells in source code can negatively impact software maintenance and evolutionary activities. Despite the willingness spent by researchers on this topic, we noticed that several previous studies consider as the "lowest common denominator" the adoption of Java programming language [102, 118, 348]. Although the use of Java is consolidated over time, other programming languages—i. e., Python—are increasingly widespread; a recent statistic[1] reports that Python jumped over Java in terms of diffusion in the last few years. Although the possible reasons for this overtaking are multiple, we noticed that it is common practice for practitioners and big companies to select programming languages that allow combining different paradigms—e. g., object-oriented and procedural— taking full advantage of the features of each of the paradigms, characteristics that are by default in Python. Moreover, from this perspective, we noticed that only a tiny subset of previous work focuses on detecting code smells for Python projects but considers only traditional systems [64, 65]. However, taking into account that Python is one of the most popular programming languages to build AI-enabled Systems[2] and considering the different philosophy in terms of the mindset of Python, we noticed a lack of empirical investigation on the diffusion of code smells in AI-enabled Systems, and the activities performed by developers during the introduction of them.

[1]https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/

[2]https://bootcamp.berkeley.edu/blog/ai-programming-languages/

Seek consolidated literature on code smells in traditional systems drove us to investigate them [98, 347]. That work emphasizes identifying the diffusion of code smells and the activities most likely to cause their introduction as a first step in keeping effort low during software maintenance [383].

To fill this gap, we investigated the diffusion of the code smells and the activities that led developers to introduce them in AI-enabled Systems. To conduct our analysis, we selected over 200 AI-enabled Systems provided by NICHE dataset [372], considered over 10,600 releases identified with PyDriller, and extracted information on code smells using PySmell.

The results indicated that: I) the code smells regarding the object-oriented principles are rarely detected, and this suggests that Python developers tend to use other reuse mechanisms to build AI-enabled Systems; II) complex List Comprehension has been observed 1465 times and is both the most present and the most long alive; III) code smells do not follow a specific and common pattern over time, but the trend seems to be project-dependent; IV) the evolutionary activities are the most common activities that can induce developers to introduce code smells.

Our work makes the following main contributions:

- A preliminary analysis of the diffusion of code smells in AI-enabled Systems;
- A preliminary investigation on the activities performed by developers that led to the introduction of code smells in AI-enabled Systems;
- A publicly available replication package [117] containing raw data and scripts used to conduct our work that researchers can use to replicate or extend this work.

6.2 RESEARCH QUESTIONS AND METHOD

The ultimate *goal* of this preliminary study is to analyze the diffusion of code smells in AI-Enabled systems and understand the activities performed by developers that, in turn, induced the introduction of code smells, with the *aim* to identify how code smells are distributed in AI-

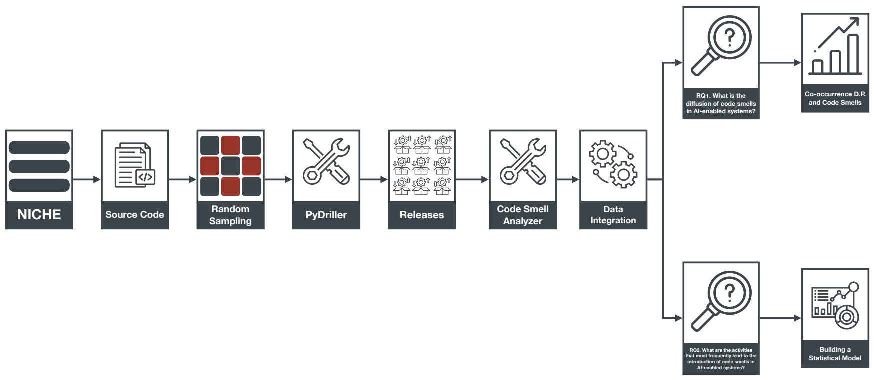
NICHE
Source Code
Random Sampling
PyDriller
Releases
Code Smell Analyzer
Data Integration
RQ1. What is the diffusion of code smells in AI-enabled systems?
Co-occurrence D.P. and Code Smells
RQ2. What are the activities that most frequently lead to the introduction of code smells in AI-enabled systems?
Building a Statistical Model

Figure 6.1: Overview of the method applied in this study.

Enabled systems, and what stages of development are most likely to introduce code smells. The *perspective* is for both developers and researchers. The former are interested in avoiding an incidental introduction of code smells that can increase the effort during maintenance and evolutionary activities. The latter are interested in enhancing the knowledge of code smells during the software evolution in systems different from Java. For this reason, we formulated the following research questions:

RQ$_1$. *What is the diffusion of code smells in AI-Enabled systems?*

RQ$_2$. *What are the activities that most frequently lead to the introduction of code smells in AI-Enabled systems?*

The *objective* of the **RQ$_1$** is to assess the diffusion of code smells in terms of *frequency*, *density*, and *variation* with the *purpose* to give a general overview of code smells in AI-Enabled systems. For these reasons, we identified three sub-research questions:

RQ$_{1.1}$. *What is the frequency of code smells in AI-Enabled systems?*

RQ$_{1.2}$. *What is the density of code smells in AI-Enabled systems?*

RQ$_{1.3}$. *What is the variation of code smells in AI-Enabled systems?*

While the *objective* of the **RQ**$_\mathbf{2}$ is to identify commits that introduced new code smells with the *purpose* of identifying which kinds of activities most frequently induce the introduction of new code smells. To conduct our experiments, we followed the empirical software engineering principles and guidelines of Wohlin et al. [376]. In addition, we follow the *ACM/SIGSOFT Empirical Standards* [3] to report our results. We include all the material, including scripts, raw data, and figures, in our online appendix publicity available [117]. Figure 6.1 provides an overview of the method applied to perform our study.

Dataset Selection

The *context* of this experiment is composed of 200 AI-specific projects and over 10,600 releases.

More specifically, to perform our analysis, we used NICHE dataset [372]—i. e., a dataset published in 2023 that contains 572 AI-specific projects. The reasons why we chose this dataset are multiple. On the one hand, the authors filter out unpopular projects—i. e., projects with less than 100 stars and no longer active projects. On the other hand, they manually verified information about the quality of the projects using a heuristic approach by labeling 400 projects as "well-engineered" according to 8 distinct dimensions: Architecture, Community, Continuous Integration, Documentation, History, Issues, License, and Unit Testing.

ARCHITECTURE. The projects have a clear definition of the components and how they communicate with other parts of the software system.

COMMUNITY. All software projects have numerous collaborators who maintain the repository.

CONTINUOUS INTEGRATION. The projects use a CI mechanism that ensures stable source code for development or release.

[3] Available at: https://github.com/acmsigsoft/EmpiricalStandards. We followed the "General Standard" and "Repository Mining"guidelines.

DOCUMENTATION. All projects provide documentation and additional material useful during maintenance activities.

HISTORY. Software systems have a long history that demonstrates developers performing maintenance and evolutionary tasks to ensure a high level of functionality.

ISSUES. The management activities have been done only using the GitHub issue, thus improving the traceability between requirements and source code.

LICENSE. Projects clearly display a usage license that is helpful for understanding the terms and conditions related to the partial or complete reuse of system components.

UNIT TEST. To ensure a good quality level, all the projects show unit tests used to verify the correctness of the component.

We focused on the 400 projects labeled as "well-engineered" and randomly selected a statistically significant sampling of 200 projects, considering a confidence level of 95%, and a margin error of 5%.

Data Collection

Once we identified the sample of projects, due to the time-consuming activity, we set up PyDriller [327] to extract only commits marked as "release" according to GitHub, and pull out the corresponding commit message. We decided to focus only on these commits because they are typically released after a more meticulous inspection by developers.[4] At the end of this step, we collected information on over 10,600 releases. To extract Python-specific code smell, we used PySmell [64]—i. e., a code smell static analyzer tool. The principal motivations that drove to use it are that: I) PySmell can detect 11 types of Python-specific code smell, and

[4]https://docs.github.com/en/repositories/releasing-projects-on-github/about-releases

the authors manually validated the instances of code smells; II) The tool is one of the most used in previous work on this topic [66, 353].

Finally, to better perform our analysis, we discarded projects not useful for our study i. e., projects with zero instances of code smells and projects with fewer than two releases. At the end of this phase, we obtained 34 projects useful for our analysis.

To the sake of comprehension, we report the list of code smells detectable by PySmell with the relative detection rule in Table 6.1.

Code Smell	Acronym	Description	Detection rule
Large Class	LG	A class with a large number of operations.	Lines of code (LOC) ≥ 200 or Number of Attributes (NOA) + Number of Methods (NOM) > 40.
Long Parameter List	LPL	A method or a function that contains a long list of parameters.	Number of Parameters (PAR) ≥ 5.
Long Method	LM	A method or a function containing many Lines of Code (LOC).	Method Lines of Code (MLOC) ≥ 100.
Long Message Chain	LMC	An expression that accesses an object using a long line of dot operations.	Length of Message Chain (LMC) ≥ 4.
Long Scope Chain	LSC	A method or a function that shows a multiple-nested.	Depth of Closure (DOC) ≥ 3.
Long Base Class List	LBCL	A class that has been defined with too many base classes	Number of Base Classes (NBC) ≥ 3.
Useless Exception Handling	UEH	An exception too many generic or that contains an empty statement.	Number of Except Clauses ((NEC)= 1 and Number of General Exception Clauses (NGEC) = 1) or NEC = Number of Empty Except Clauses (NEEC)
Long Lambda Function	LLF	A lambda function that contains multiple and complex expressions.	Number of Characters in One Expression (NOC) ≥ 80.
Complex List Comprehension	CLC	A list comprehension that contains multiple and complex expressions.	Number of Loops (NOL) + Number of Control Conditions (NOCC) ≥ 4.
Long Element Chain	LEC	An expression accessing an object using a long list of bracket operators.	Length of Element Chain (LEC) ≥ 3.
Long Ternary Conditional Expression	LTCE	A ternary conditional expression too many long.	Number of Characters in One Expression (NOC) ≥ 40.

Table 6.1: Code Smells detectable by PySmell with the related detection rule.

Data Analysis

Once we had the data collection, we analyzed the information from both quantitative and qualitative standpoints.

To address $\mathbf{RQ_1}$, we analyzed code smell diffusion in terms of *frequency*, *density*, and *variation* over time. To analyze the *frequency*, we built a Python script to count for each release the instances of code smells, and then, we aggregated results independently from the project to provide a generic overall. To identify the *density*, we calculated the ratio between the number of smells and lines of code (LOC) for each release for all projects. Lastly, we clustered results in a time interval to calculate the *variation*.

To address $\mathbf{RQ_2}$, we analyzed the activities that led developers to introduce code smells. To address this, we performed the following steps: 1) We merged in a single *CSV* file all the output files; 2) For each pair release R_i, $R_{i+1} \in$ project P_j, we labeled the release R_{i+1} as "*increase*" if the difference in terms of the number of code smells between the version R_{i+1} and R_i is more than 0; "*stable*", in the difference equal to 0; and lastly, "*decrease*" if the difference is lower than 0; 3) To identify what activities have been done by developers who have introduced code smells we labeled the commit marked as "*increase*" with "Bug fixing ", "Evolutionary Activity", "Refactoring", or "Other" according to the corresponding commit message, as also done in previous work [347] by using a manual "pattern-matching" strategy—i. e., we manually verify the presence of specific keywords, e. g., " bug fix" to indicate a bug fixing activity—in the commit message. To perform this step, the first two authors of this work independently labeled each commit marked as "*increase*" based on what they felt was the category that corresponded to the most appropriate activity, and in case of discordance, were discussed by involving the other authors of the study until convergence was reached.

At the end of this step, all authors agreed on the assigned categories. Table 6.2 shows the labels with the relative descriptions.

Two aspects are worth discussing. First, due to ambiguous commit messages, we decided to discard from our analysis commits labeled as "Other"

Label	Description
Bug Fixing	A commit removes a bug in the source code
Evolutionary Activity	A commit that introduces a new feature in the system
Refactoring	A commit that performs a refactoring activity
Other	A commit that does not provide sufficient information to be labeled

Table 6.2: Description of the labels used.

to avoid possible noise—e. g., message commits not written in English. Second, we give, in some cases, a combination of two or more labels—e. g., Bug-Fixing and Refactoring—because, in some cases, the commit messages referred to more than one activity.

6.3 ANALYSIS OF THE RESULTS

In this section, we report the main results of our analysis and discuss findings and implications.

RQ1$_1$. On the frequency of Python-specific code smell

To address the *RQ1$_1$*, we analyze the frequency of Python-specific code smell. Figure 6.2 indicated the frequency of code smells.

According to our results, it is possible to make several considerations. First, we noticed that 3 of the 11 code smells categories were not detectable during our analysis—i. e., *Large Class*, *Long Base Class List*, and *Useless Exception Handling*. Considering that previous work on object-oriented languages underlines the predominance of these smells [319] and that all of them referred to improper use of the object-oriented principle, the main assumption of the absence of this family of smells is that developers do not adopt or only partially adopt object-oriented approaches to build AI-enabled systems, but prefer others reuse strategies. Second, we noticed a clear gap between the first two smells—i. e., *Complex List Comprehension* and *Long Ternary Conditional Expression*—which appear respectively 1400 and 226 times and the other 6 smells. In both cases, the smells refer to

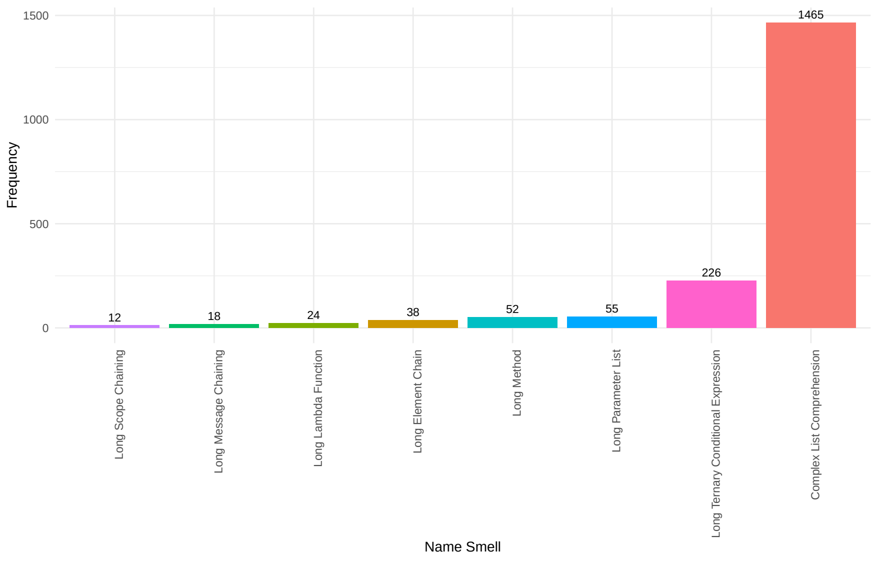

Figure 6.2: Results of the frequency of Python-specific code smell over time.

syntactic contractions to reduce the lines of code required to perform an operation. This result suggests a possible correlation between the Python philosophy that encourages developers to write compact code snippets and the massive presence of these smells.

$RQ1_2$. On the density of Python-specific code smell

To address the *$RQ1_2$* we analyze the density of code smells from an evolutionary perspective. We observed that they often do not exhibit a consistent pattern of increase/decrease over time. Instead, they appear to be influenced by external factors, as exemplified by the anomaly observed in row 4, column 6, where an unstable pattern can be observed. These anomalies lead us to believe that the code smells introduction and their removal could vary causally due to software evolution activities. Figure 6.3 provides the density overview for all the projects under analysis.

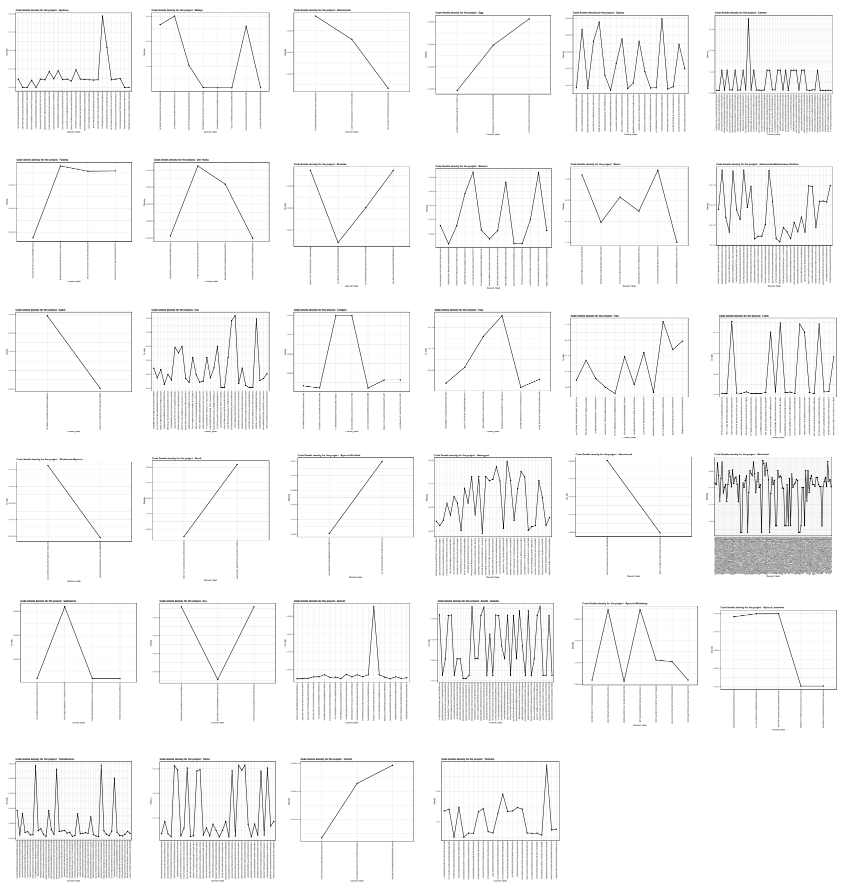

Figure 6.3: Overview of the density of code smells for each project analyzed.

RQ1$_3$. *On the variation of Python-specific code smell*

To address the *RQ1*$_3$ we analyzed the code smells variation. We decided to show the results in a 3-month interval for readability reasons.

Figure 6.4 shows the code smells trend over time. Several considerations can be made when looking at the figure. First, no common pattern has been identified, suggesting that the code smell variation also seems project-dependent. Perhaps more interesting, we noticed that 80% of the projects had been affected at least once by a *Complex List Comprehension.* This

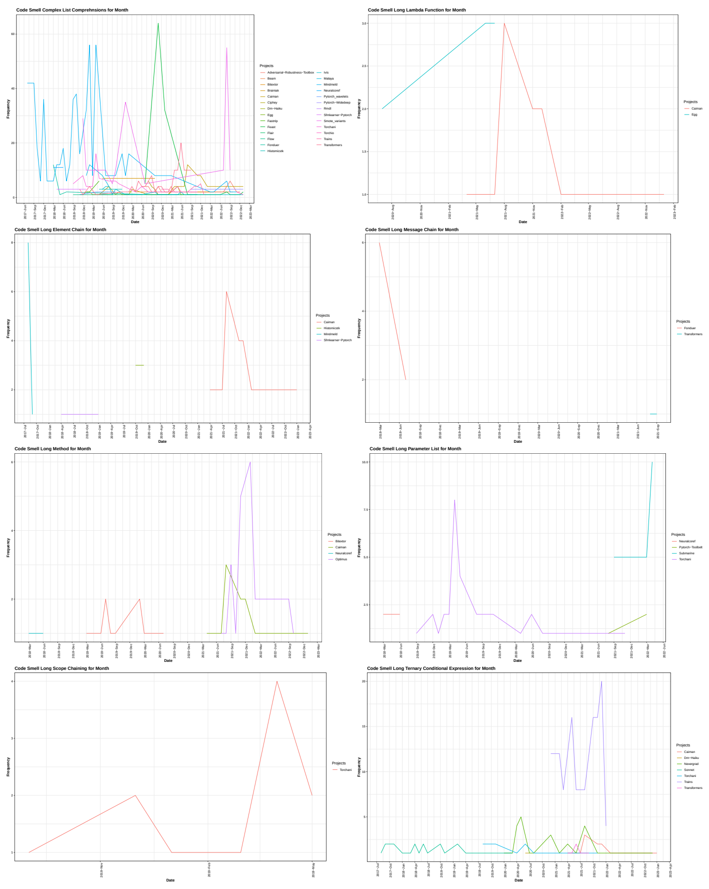

Figure 6.4: Results of the variation of Python-specific code smell per month.

outcome reinforces the results of RQ1_1, showing that introducing this kind of smell is frequent in these systems. Lastly, we noticed that this smell is also one of the longest-lived, as in some cases, its presence covers a period from 2017 to 2023.

Key findings of RQ_1.

The density of code smells does not follow a specific pattern but varies depending on the project being considered. Code smells related to object-oriented practices are never detected during our analysis. The most frequent smell is *Complex List Comprehension*, with 1465 observations that are also the longest-lived.

RQ_2. On the activities that led developers to introduce code smells in AI-enabled Systems

To address the RQ_2, we analyzed activities that led developers to introduce code smells in AI-enabled Systems as specified in the section 6.2.

Figure 6.5 shows the results obtained.

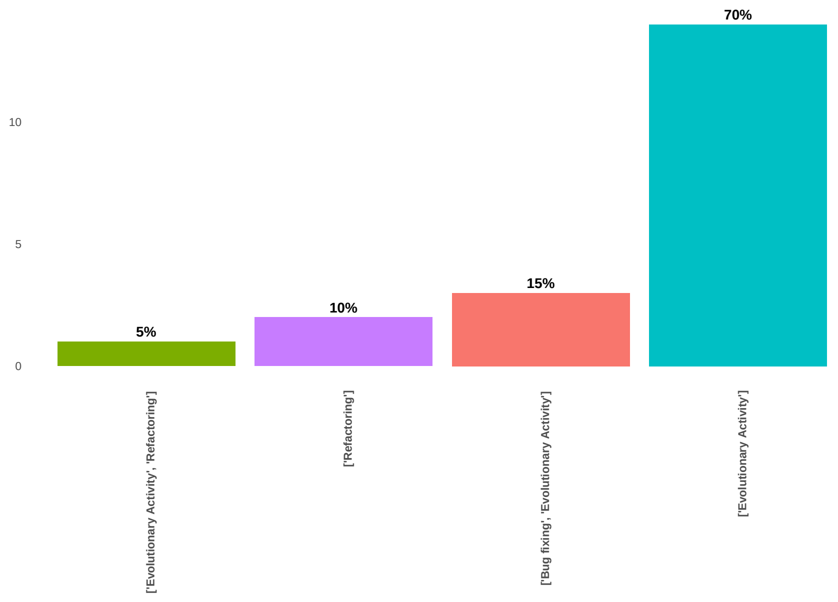

Figure 6.5: Activities performed by developers during the introduction of code smells.

The main outcome of this RQ is that developers introduce code smells in 70% of the cases during evolutionary activities. More in detail, by analyzing the commit messages it was observed that in most cases the commits, were related to merge activities or had generic messages indicating a system update. Although no further clarification can be provided for those generic commit messages related to system upgrades, it is assumed that the complexity and criticality of merge activities increase the likelihood of introducing code smells. Furthermore, due to the unstable trend exhibited over time, we can assume the developers tend to neglect the adoption of *quality assurance* tools throughout the software life cycle for monitoring code quality attributes. This highlights an unawareness regarding the potential impact of software quality degradation, which can lead to increased system complexity and the introduction of software bugs.

Key findings of RQ$_2$.

Code smell introduction is most common in evolutionary activities. In particular, we noticed that the merge operations could drastically increase the possibility of introducing them in AI-enabled Systems. Finally, our findings suggest a lack of awareness by practitioners of the importance of monitoring quality attributes of source code as their systems evolve.

6.4 THREATS TO VALIDITY

In this section, we discuss threats to the validity that could have affected the results and the strategies we applied to mitigate them.

CONSTRUCTION VALIDITY. This threat regards the relationship between theory and observation. The crucial aspect in our case regards the dataset exploited. We are conscious that the project selection can influence the results obtained. However, to mitigate this aspect, we selected NICHE dataset—i. e., a dataset manually labeled and validated by other researchers as "well engineered" projects according to 8 different dimensions. Another threat regarding the data collection phase: to mitigate this aspect, we used well-established tools—i. e., PyDriller and PySmell. The first has been used to extract information on releases, while the second has to extract information on Python-specific code smells. In any case, we made all script, additional material, and row data publicly available for the sake of verifiability. While we recognize possible limitations of these two tools, they represent the state of the art.

CONCLUSION VALIDITY. The main threat that can affect the conclusion validity refers to the use of PySmell to detect Python-specific code smells for AI-Enabled systems. While previous research underlines that other tools cannot work in AI-Enabled systems, no studies have been performed on PySmell. As part of our agenda, we will investigate the precision and recall of this tool on AI-Enabled systems.

EXTERNAL VALIDITY. This threat is mainly connected with the generalizability of results. To mitigate this aspect, we analyzed 200 projects and 10,600 releases of open-source projects with different domains and different characteristics in terms of size, number of classes, and so on. Furthermore, we planned to conduct further analysis by increasing the number of projects and commits to assess our preliminary results.

PART III

FURTHER STUDIES ON CODE QUALITY CONSIDERING EMERGING TECHNOLOGIES

7 BACKGROUND AND RELATED WORK

This section provides the background information useful to comprehend code quality aspects considered in our secondary studies i. e., security and privacy for systems that use emerging technologies.

More in detail, section 7.1 provides a general overview of the architecture of IoT systems. Section 7.2 and section 7.3 respectively discuss the state-of-the-art of the closest studies on other SLRs on how AI was used to deal with privacy in IoT systems and previous work on the capabilities of static vulnerabilities tools was used in the context of mobile applications.

7.1 AN OVERVIEW OF IOT ARCHITECTURES

Figure 7.1 shows the current theoretical architecture of a generic IoT device. The architecture is componsed by: *perception layer* that include objects and sensors that compose the device, like GPS sensors, bar-code scanners, and RFID sensors. The *"Transport"* layer that receives pre-elaborated information and analyzes it through two sub-layers: The *Network Capabilities* and the *Transport Capabilities*. The former allows the device to connect to the network and proceed with the authentication and the access control mechanism; the latter implements the mechanisms needed to transfer the data to the upper levels (e.g., through the definition of wired or wireless protocols like Wi-Fi, RFID sensors, and Bluetooth. The *"Processing"* layer offers a support mechanism to store the data received by the lower layer; this layer usually uses cloud infrastructures, ubiquitous computing, and, finally, it is used to perform data analysis tasks and generate actions that could influence the environment. From the user's perspective, the *"Application"* layer offers an interface between the IoT devices and applications that could be built. It is used to develop and deploy IoT applications like a

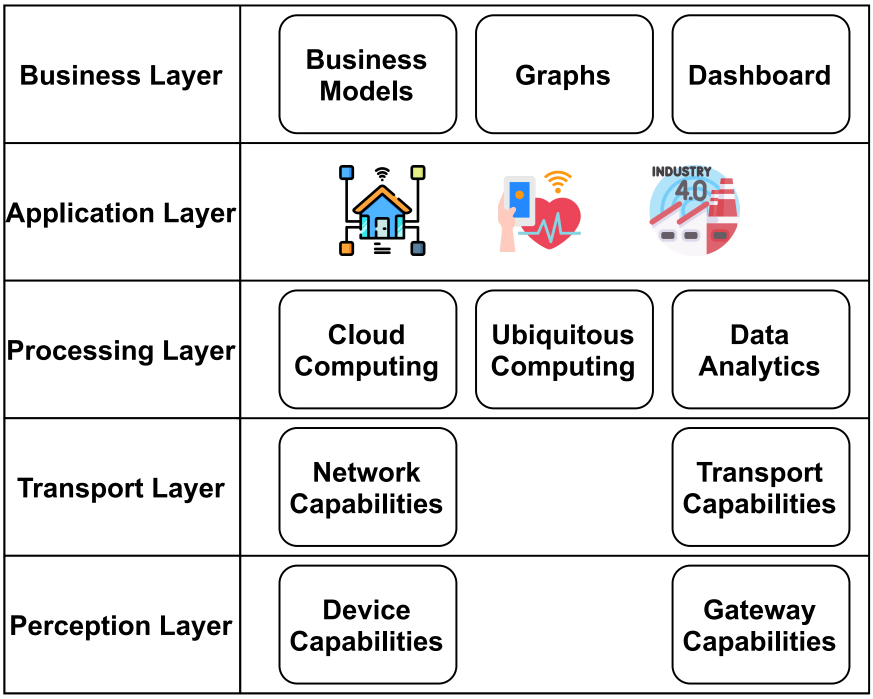

Figure 7.1: Five Layer Architecture.

specific application to govern a smart home or monitor a patient with a healthcare app. Finally, the *"Business"* layer is used to manage and control applications using flow charts, graphs, and dashboards. This is the layer employed for the decision-making process, where one can decide which actions or operations should be done with the information received from the previous layers. This layer is directly involved in protecting the end user's privacy. In any case, it is worth remarking that, in a real-world scenario, the architectures described are often subject to changes or customization to meet specific requirements of the application to build or because of the heterogeneity of the IoT devices. As such, the architectures should be considered as a starting point for building IoT applications.

7.2 A LITERATURE REVIEW OF SECURITY AND PRIVACY IN IOT SYSTEMS

Recently, some literature reviews targeted the privacy of IoT systems. Most of them treated the problem by investigating the major privacy threats in IoT environments, focusing on the root causes of privacy concerns rather than how artificial intelligence methods have been exploited to identify privacy issues or preserve privacy.

Aleisa and Renaud [11] surveyed the literature from 2009 and 2016 to investigate (1) the geographic distribution of privacy issues, finding this typically concerns Europe and North America; (2) the data collection methods, which were found to be diverse and scattered, other than mainly focusing on quantitative perspective; (3) the hardware technologies, that in about 35% of the cases refer to RFID sensors; (4) the major issue about privacy, namely the lack of privacy-preserving mechanisms; and (5) the topics treated, with authentication and authorization mechanisms being the most popular ones. In doing so, the authors did not only collect published research papers but also news stories and privacy reports to analyze a larger variety of privacy violation perspectives.

Ziegeldorf *et al.* [410] elaborated on a list of IoT environments' privacy threats, reporting the following ones as the most harmful:

IDENTIFICATION. This relates to the possibility of identifying a device through its IP address or machine name;

LOCALIZATION AND TRACKING. This threat refers to the possibility of detecting user traffic in multiple ways, e. g., using a GPS sensor or smartphone localization;

PROFILING. The profiling threat involves the potential for tracking user information to identify relevant data about the target;

INTERACTION AND PRESENTATION. This aspect refers to Machine-Machine Interaction. Indeed, a threat to privacy could arise when these devices share information with other devices;

LIFECYCLE TRANSITIONS. This threat occurs when the devices assume that the information previously shared with other devices has been deleted. However, the devices that receive that information could be storing those data for unclear reasons;

INVENTORY ATTACKS. This aspect refers to the possible unauthorized access inside the device. Indeed, the malicious user could detect possible sensible data and use it for multiple illegal actions;

LINKAGE. This threat refers to privacy issues arising when multiple devices are connected and share information; in these cases, the devices could be used for unauthorized access inside the system.

The work by Ziegeldorf *et al.* [410] is not meant to be a systematic investigation but rather a viewpoint on the key concerns threatening the privacy of IoT systems.

Two literature reviews have been recently published by Hussain *et al.* [152] and Waheed *et al.* [360]. Similarly to us, both of them investigated the role of artificial intelligence in the context of IoT privacy.

Hussain *et al.* [152] conducted a literature review to delineate the current solutions and the future challenges of the use of machine learning in IoT environments, with a particular focus on privacy issues. From a technical standpoint, the authors conducted a meta-analysis of the previous surveys on software security and IoT systems in order to investigate two aspects. First, they synthesized the motivations for using machine learning techniques in the context of IoT. Secondly, they summarized which are the machine learning algorithms employed. In this respect, their focus was mainly on the analysis of the *efficiency* and *complexity* of the machine learning solutions proposed so far. Hence, with respect to the work by Hussain *et al.* [152], ours can be seen as a systematic and complementary analysis where we focus on the *design* of the machine learning pipelines, namely the strategies employed to train, build, and validate the models. Furthermore, our systematic literature review (I) does not limit itself to machine learning but explores the broader application of artificial intelligence methods and (II) considers additional dimensions such as the domains and the privacy issues.

Waheed *et al.* [360] conducted a systematic literature review of the research papers published from 2008 to 2019 that focused on understanding the role of machine learning and blockchain to deal with security and privacy in IoT systems. Waheed *et al.* focused on threats and countermeasures for security and privacy concerns, reporting the lack of survey efforts in the context of machine learning and privacy.

7.3 LITERATURE REVIEW ON ANDROID VULNERABILITIES

A significant amount of previous work designed automated techniques to identify vulnerabilities: they establish the level of security of mobile apps with respect to various vulnerability types based on the analysis of various static constructs [303], textual analysis [83], or data flow (e.g., use-def relations [71]). A systematic overview of these approaches has been recently proposed by Li et al. [197].

A number of studies focused on understanding ANDROID vulnerabilities. Linares-Vásquez et al. [205] classified the vulnerabilities affecting the ANDROID OS. In contrast, Gao et al. [109] focused on the evolution of software vulnerabilities from the perspective of mobile apps. Additional studies pertained to third-party libraries [296, 393] and how they might potentially threaten software security aspects of source code.

A few taxonomies of software vulnerabilities in mobile apps have been proposed. Sadeghi et al. [293] conducted a systematic literature review on the research on mobile app security, defining a taxonomy of the vulnerabilities treated by researchers over the years. Qamar et al. [284] and Mirza et al. [242] defined context-specific taxonomies that cover the vulnerabilities affecting the mobile banking domain.

8 ON THE USE OF ARTIFICIAL INTELLIGENCE TO DEAL WITH PRIVACY IN IOT SYSTEMS: A SYSTEMATIC LITERATURE REVIEW

8.1 INTRODUCTION

We live in a world that is more and more virtualized and where people can do anything, anytime, from anywhere [341].This is enabled by the availability of devices and sensors that can capture the surrounding environment and/or the user requests in order to distribute them toward other devices and sensors and produce data, knowledge, actions, communications, entertainment, and others: this is what we call *Internet-of-Things* (a.k.a. IoT) [33]. It is not easy to give an all-encompassing definition of IoT because the field of applicability of these devices is so vast to risk not including some possible domain or sub-domain. However, Strous *et al.* [331] tried to give a general definition that can be summarized as *"IoT is the inter-networking of physical devices such as vehicles, home appliances, medical devices and so on that can collect and exchange data and interact with other devices using the Internet to monitor or control something."* The key objective of IoT is indeed that of providing people with an infrastructure that allows ubiquitous access to devices and service providers [204]. The last decades have seen an ever-growing interest in IoT, and the vast majority of services and communications are currently offered through IoT devices like smartphones and other smart objects [129, 177]. Recent statistics report that, in 2020, the number of IoT devices connected to the Internet is about 8.74 billion, and this number will increase by 25 times in 2030.[1] However, the growth of IoT is not exempt from serious security threats and privacy [410].

[1]Source STATISTA.COM: `https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/`.

In particular, IoT devices typically have low memory and produce large amounts of sensitive data that are sent and elaborated by servers, which then return the outcome of the elaboration to those devices [168]. Such an intensive exchange of data naturally allows external attackers to steal information and use them for malicious reasons [13, 258], as we witness too often in the news. Some of the most recent, resounding examples are connected to the malicious use of smart assistants[2] or even the influence that IoT data leaks might have had on the 2016 US elections.[3] The problem of privacy is so spread in practice that Meneghello *et al.* [236] defined IoT as the *Internet-of-Threats*, synthesizing the current body of knowledge on security weaknesses of commercial IoT solutions and highlighting the need for automated mechanisms that may support the detection of privacy concerns in IoT systems. Researchers have actively embraced this call through the definition of techniques based on blockchain [186, 187], gateway instrumentation [231], privacy-preserving data aggregation schemas [195, 212], to name a few.

Besides the techniques discussed above, a recent trend is represented by the adoption of artificial intelligence (AI) algorithms and models. These approaches concern the design of supervised and unsupervised methods, meta-heuristics, or reasoning approaches to detect potential privacy leaks or to preserve privacy in IoT systems [247]. For example, Majumder and Izaguirre [222] developed an AI-based security system that, employing motion detection and facial recognition, might prevent the malicious intrusion of externals into IoT systems. Similarly, Liu *et al.* [208] developed a fully encrypted *Convolutional Neural Network* (CNN) [10] to monitor the vital signs of patients: the encryption mechanism allowed to hide personal data during the training phase of the artificial intelligence model, preserving privacy.

Most of this research has been conducted by researchers in the fields of algorithms, cybersecurity [70], and networks. We advocate that it is time

[2]The ALEXA case: https://www.theguardian.com/technology/2019/oct/09/alexa-are-you-invading-my-privacy-the-dark-side-of-our-voice-assistants

[3]The CAMBRIDGE ANALYTICA case: https://www.theguardian.com/news/2018/mar/17/cambridge-analytica-facebook-influence-us-election.

for software engineering to come into play by conducting empirical investigations into the matter and proposing novel instruments to support developers of IoT systems. In this respect, we notice a lack of comprehensive knowledge on what are the privacy issues tackled by artificial intelligence approaches, which design and validation choices were applied when assessing those techniques, which are the current limitations induced by these choices, and in which domains the application of AI-based methods seem more promising. An improved understanding of these aspects is crucial for (I) assessing the current capabilities of these methods; (II) pointing out potential limitations of the techniques employed so far; and (II) identifying additional domains and/or methods that may be used to detect possible privacy issues and preserve data privacy in IoT systems. All these angles might be of interest to our research community to identify the areas where to focus our collective effort.

Hence, in this chapter, we conduct a Systematic Literature Review (SLR) on the usage of artificial intelligence techniques for the detection of privacy issues or to preserve privacy in IoT systems.

We employ well-established guidelines [175, 374] to systematically search the literature on the matter: from an initial set of 2,202 papers, we identify 152 primary studies that we then analyze to address the three perspectives of interest.

The analysis of the research literature highlights an increasing interest in artificial intelligence methods for the privacy of IoT systems, and, indeed, we find that a large portion of the papers were published in 2020. In addition, we identify two key use cases: artificial intelligence is used to spot privacy issues or prevent their emergence, yet there exist several sub-fields where AI-based techniques might be applied. While the most widely used approach is *Support Vector Machine*, we discover that only a few papers elaborated on the rationale behind the selection of the AI technique and, perhaps more importantly, that most of the approaches have been assessed through a limited and potentially biased evaluation metrics. Lastly, we find that the vast majority of the published papers do not include explicit indications on the domains where the proposed techniques can be applied, hence threatening their actionability and reproducibility.

Based on our findings, we identify several future research directions and implications for the research community that encompass the adoption, definition, configuration, and validation of artificial intelligence methods for IoT privacy.

8.2 RESEARCH QUESTIONS AND METHOD

The *goal* of the study is to survey the research literature that applied artificial intelligence methods for detecting privacy concerns and preserving privacy in IoT systems, with the *purpose* of providing software engineering researchers with actionable items and insights that they can exploit to investigate the matter further and improve the automated support made available to developers and managers to deal with privacy concerns. The *perspective* is that of researchers who are interested in assessing the currently existing methodologies and how to improve them.

To address our goal, we developed and conducted a Systematic Literature Review (SLR), which is a synthesis process through which the existing research papers on a subject of interest are systematically identified, selected, and critically appraised to address one or more research questions [175]. In the context of our work, we followed the well-established guidelines originally proposed by Kitchenham and Charters [175]. To provide additional rigor to the analysis, we also integrated the standard procedure with the so-called *snowballing* procedure [375], i.e., a methodology used to scan the incoming and outcoming references of the primary studies identified by the systematic search for identifying additional sources. We followed the snowballing guidelines provided by Wohlin [374]. In terms of reporting, we followed the *ACM/SIGSOFT Empirical Standards*.[4] and, in particular, the *"General Standard"* and *"Systematic Reviews"* guidelines.

[4]Available at: `shorturl.at/cBDH6`.

Research Objectives and Questions

The specific research objectives of the systematic literature review are reported in the following:

OBJECTIVE 1. Understanding the IoT privacy tasks targeted with artificial intelligence techniques;

OBJECTIVE 2. Understanding the IoT domains where artificial intelligence techniques have been employed.

OBJECTIVE 3. Understanding the design, configuration, and evaluation of AI techniques for privacy in IoT systems.

While the literature on privacy of IoT systems [342] has established a number of static and dynamic instruments that help developers detecting the presence of privacy threats, our objectives are motivated by recent research efforts in the field of privacy and security showing an increasing trend in the adoption of artificial intelligence methods to deal privacy in IoT systems. [181, 245]. For instance, Kuzlu *et al.* [181] advocated the exponential growth in the development of complex artificial intelligence-enabled algorithms to protect IoT systems. This was confirmed by a number of additional studies in the field of privacy and security (e.g., [115, 261, 378]). These observations posed the foundations of our research objectives. We argue the need for a comprehensive understanding of how artificial intelligence methods have been engineered for securing the privacy of IoT systems. This is crucial to assess the software engineering angle of the matter, possibly letting emerge problems and challenges that our research community might help addressing.

Our objectives have influenced the formulation of our **RQs**:

🔍 **RQ$_1$.** *What are the IoT privacy tasks that can be tackled with the use of artificial intelligence techniques?*

RQ$_1$ aimed at addressing the first objective and investigating the most common privacy tasks addressed through the adoption of any form of artificial intelligence method. The research question is motivated by our willingness to provide a comprehensive overview of the state of art regarding privacy tasks treatable with AI-based methods: this may reveal tasks that have been only partially considered by the state of the art, hence suggesting potential future work in the field.

🔍 **RQ**$_2$. *Which are the IoT domains where artificial intelligence techniques have been applied to deal with privacy?*

This research question addressed the second overall objective of the study and was motivated by the willingness to assess the typical domains where artificial intelligence techniques have been applied to the problem of privacy, which may naturally highlight additional domains where the application of these techniques might be worthy.

🔍 **RQ**$_3$. *Which families of artificial intelligence algorithms were used to deal with privacy in IoT systems?*

🔍 **RQ**$_4$. *Which datasets were used to validate the artificial intelligence methods employed to deal with privacy in IoT systems?*

🔍 **RQ**$_5$. *Which strategies were used to validate the artificial intelligence methods employed to deal with privacy in IoT systems?*

🔍 **RQ**$_6$. *What are the evaluation metrics employed to assess the quality of the artificial intelligence methods employed to deal with privacy in IoT systems?*

With the set of research questions from **RQ**$_3$ to **RQ**$_6$, we aimed at addressing the last objective and investigating the inner-working of the artificial intelligence techniques employed in the literature. It is important to note that such an analysis was motivated by a key consideration in the field of artificial intelligence and machine learning: the design, configuration, and validation of those techniques might heavily influence the interpretation of the performance [221, 254, 301]. Hence, our research questions shed light on how researchers have defined these techniques, possibly revealing common patterns and limitations to address. In addition, these research angles allowed us to complement previous work on the use of artificial intelligence methods for IoT privacy [152, 360], by providing a deeper understanding of the methodology used to define artificial intelligence pipelines to be used when detecting privacy issues or preserving privacy.

Research Query definition

One of the key methodological steps of a systematic literature review is identifying appropriate search terms that may help retrieve a comprehensive set of sources. In this respect, we adopted the following strategy:

- For each research question, we first extracted the most relevant keywords—these represented the base to conduct our search;
- For all relevant terms, we identified possible synonymous or alternative spelling;
- We used boolean operators to compose the research query.

The outcome was the following research query:

Search Query

(“privacy” *OR* “anonymization” *OR* “sensitive information” *OR* “sensitive words”) *AND* (“iot” *OR* “Internet-of-Things”) *AND* (“machine learning” *OR* “artificial intelligence”)

As shown, we put in *OR* all the synonyms of the same concept, while multiple concepts were combined using the *AND* operator. The basic idea behind this step was to verify the consistency and completeness of the selected terms against papers that reported a systematic investigation of the literature. Therefore, they are supposed to contain a complete mapping of the terms used in literature to indicate privacy concerns. This step allowed us to include alternative words in the search query in case these were not included initially. This did not eventually happen since we did not identify any additional terms to include.

As for the artificial intelligence-related terms, the existing literature reviews targeted just part of the problem, i.e., the use of machine learning. Therefore, it was impossible to check the selected terms' completeness against them. In any case, we preferred to include both *"machine learning"* and *"artificial intelligence"* to (1) not miss any of the resources considered by previous systematic literature review and (2) identify sources that did not employ machine learning but other forms of artificial intelligence.

Search Databases

Once we defined the search query, we selected the databases to use when performing our search. Correct identification of those databases is fundamental to have a successful literature review [175]. For this reason, we selected the top-three research databases,[5] namely:

- IEEEXplore (http://ieeexplore.ieee.org);
- Scopus (www.scopus.com);
- ACM Digital Library (https://dl.acm.org).

These databases are typically used to conduct systematic literature reviews [47, 174] and, perhaps more importantly, guarantee complete coverage of the published research, hence allowing us to access the entire set of papers.

[5]source: `https://paperpile.com/g/research-databases-computer-science/`.

Exclusion and Inclusion criteria

Exclusion and inclusion criteria allow the selection of resources that address the research questions of a systematic literature review [174]. In the context of our study, we identified and applied the following *"Inclusion/Exclusion"* criteria.

EXCLUSION CRITERIA: The resources that met the following constraints were filtered out from our study:

- Papers not written in English;
- Short papers, namely papers with a number of pages lower than seven;
- Workshop papers;
- Duplicated papers;
- Papers whose full text read was not available;
- Conference papers later extended to journal;
- Master Theses.

Using these filters, we could exclude all preliminary research results, e.g., workshop or short papers, but also avoid considering a similar paper multiple times, e.g., in case of an archived journal paper that extends a conference publication or in case of duplicates.

INCLUSION CRITERIA: Papers that applied artificial intelligence methods to the problem of privacy of IoT systems were *included* in our study.

Snowballing

The snowballing technique refers to the use of the reference list of an paper or its citations to identify additional papers that might have been missed by the search process [375]. This is typically used *after* the application of the exclusion/inclusion criteria, so that the reference analysis is only performed on the relevant papers that address the research questions of the literature review. As the reader might see, the snowballing technique requires an extensive amount of time and effort: for this reason, we limited ourselves to the application of the so-called *backward* snowballing, that is, the scanning of the reference list of the papers selected.

Quality assessment

Before proceeding with the extraction of the data required to address our research questions, we assessed the quality and thoroughness of the retrieved resources to discard the papers that did not provide enough details to be used in our study. Particularly, we defined a checklist that included the following questions:

Q1. *Are the artificial intelligence techniques clearly defined?*

Q2. *Are the privacy topics treated in the paper clearly defined?*

Each question could be answered as *"Yes"*, *"Partially"*, *"No"*. We associated a numeric value for each label better to assess the quality and thoroughness of each source: the label *"Yes"* was associated to the value *'1'*, *"Partially"* to *'0.5'*, *"No"* to *'0'*. The overall quality score was computed by summing up the score of the answers to the two questions, and the articles with a quality score of at least 1 were accepted.

Data extraction

Once we had identified the set of sources to consider, we extracted the information relevant to address our research questions. We defined the data extraction form reported in Table 8.1. Besides the basic information on the privacy topics treated by the paper or on the design/validation of the artificial intelligence techniques, we also sought to extract data on the dataset exploited and the programming language used to develop the technique: these pieces of information could provide additional insights into the characteristics of the considered papers. Also, the data extraction

Table 8.1: Data Extraction Form

Dimension	Attribute: Description
Privacy	What kind of privacy concerns have been highlighted in the use of IoT devices?
Machine Learning Algorithms	What kind of algorithms were used to tackle the problem?
Topics of Interest	What are the main topics treated?
Programming Language	What programming languages have been used to address this issue?
Training Strategy	What is the strategy adopted to train the model?
Validation Techniques	What kind of techniques were used to validate the model (if any)?
Dataset	What dataset has been selected to train the Machine Learning model?
Evaluation Metrics	What evaluation metrics has been used to evaluate the model? (e.g., F-score, Accuracy, Precision, Recall).
Limitation	What are the limitations of current techniques?

form included a *"Limitation(s)"* field, through which we took note of the possible limitations of the techniques assessed. It is important to remark that the *"Validation Techniques"* field of the data extraction form was voluntarily left optional, as not all the primary studies might have proposed validations of the artificial intelligence techniques proposed.

Search Process Execution

Once we had defined the basic blocks of our systematic literature review, we then proceeded with its execution. An overview of the execution is presented in Figure 8.1, where we show how the number of primary studies considered varied when applying the various filters we defined. In particular, the execution process worked as follows:

Table 8.2: Filters applied in the research queries.

Database	Year	Document Type	Publication Stage	Language	Media Format	Subject Area	Source Type	Results
IEEE	2011 - 2021	Conference Journal						486
Scopus	2011 - 2021	Conferences paper	Final	EN		Computer Science Engineering	Conference Proceeding Journal	443
ACM	2011 - 2021	Research Paper			PDF			1,273
Total								2,202

A We run the search query against the three selected databases. In this respect, it is worth remarking that each database requires its parameters to conduct the search process, e. g., in terms of the document types to consider. For the sake of replicability, Table 8.2 summarizes the parameters defined for each database. The search query output

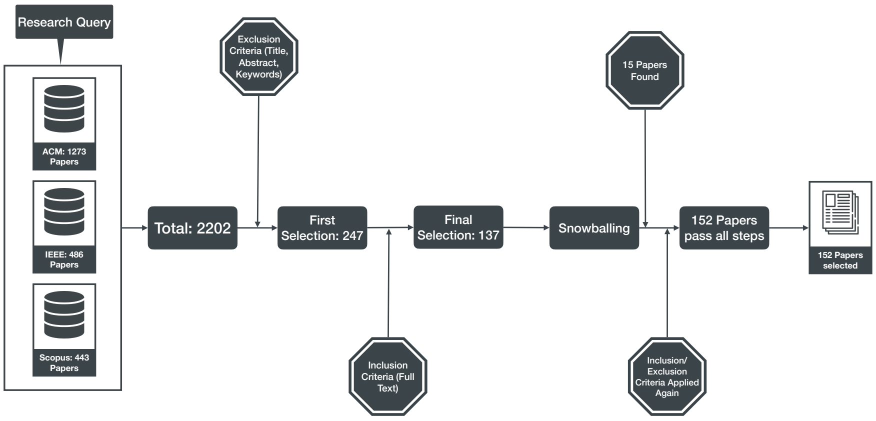

Figure 8.1: Overview of the papers selection process.

a total amount of 2,202 hits: 1,273 for the ACM Digital Library, 486 for IEEEXplore, and 443 for Scopus. The higher number of hits obtained when querying the ACM Digital Library is motivated by the internal mechanisms that the platform employs to match a query against the content it makes available [42]: in particular, it does not limit the search of each term of a query to the full content of an paper, but also considers the metadata, hence providing a larger amount of candidate relevant papers. In any case, we completed the first step by downloading all the candidate papers and storing them in a local environment for a quicker investigation.

B Each of the candidate papers entered the next phase, which consisted of applying the exclusion criteria. We considered each paper's title, abstract, and keywords to decide on whether it should have been discarded. If this is enough, we read the content of the paper. Overall, 1,955 papers were excluded, and, therefore, 247 passed to the next step.

C The inclusion criteria were considered. Also, in this case, the paper's first author acted as the inspector and applied the criteria defined against the 247 papers. Unlike the previous step, the inclusion was assessed by considering the full paper and not only the title, abstract, and keywords. In case of indecision, the inspector brought the case to the attention of the other authors, who could provide feedback and open a discussion that led to a final agreement. As a result, we discarded an additional 110 sources, leading to a final number of 137 papers included in our systematic review.

D After collecting the relevant papers, the inspector applied the backward snowballing procedure and identified potentially relevant candidates missed by the original search. Then, the inspector let the additional sources pass through the exclusion/inclusion criteria. Similar to what was previously done, the inspector requested the feedback of the other authors whenever needed. The snowballing procedure included 15 new sources, leading to a total of 152 papers.

E The next step is concerned with the application of the quality assessment. This was a critical phase since we had to rate the papers based on their clarity or the availability of enough information to address our research questions. As a result, no paper was excluded, and, therefore, all 152 papers passed the quality assessment.

F Lastly, we proceeded with the data extraction. Most of the information required to address our research questions (i. e., , the AI technique employed) was rather easy to collect. More problematic was instead the analysis of the potential limitations. This required a more careful and focused discussion. In particular, we analyzed (I) the sections of the papers where potential limitations and threats to validity were discussed and (II) the characteristics and properties of each technique employed, trying to identify additional limitations.

The data extracted from the selected papers were then used to provide an answer to our research questions. The following section overviews the main findings of our analysis.

8.3 ANALYSIS OF THE RESULTS

Before diving into the results addressing our **RQ**s, it is worth reporting some meta-information on the primary studies accepted for our systematic literature review.

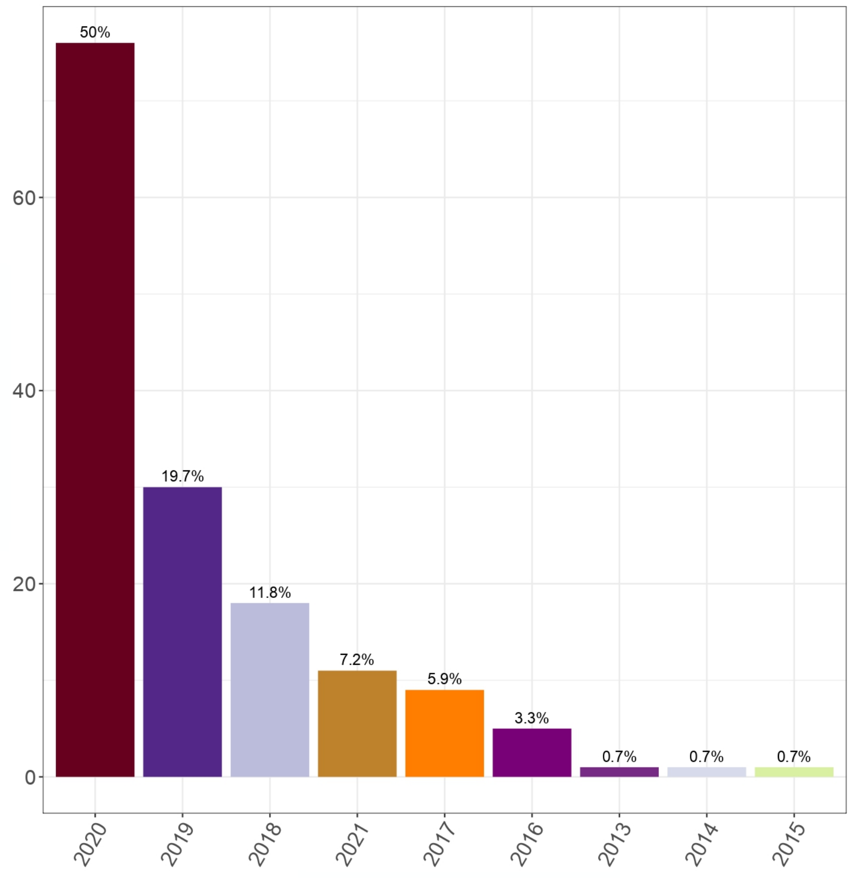

Figure 8.2: Publication trend by year.

In the first place, Figure 8.2 depicts bar plots highlighting the number of papers published by year. Looking at the figure, two elements might be noticed. On the one hand, the publication trend recalls an exponential function, which means that the interest in facing privacy in IoT systems using artificial intelligence techniques is rapidly and massively increasing. This may indicate that several other papers will be published in the near future. The publication trend further motivates our work, namely the need for a systematic literature review that analyzes how artificial intelligence techniques have been applied and validated in the field, other than which are the key limitations that future research is called to address.

On the other hand, we can also notice that 50% of the papers (75) have been published in 2020 [1, 3–5, 15, 18, 29–31, 40, 41, 43, 50, 54, 55, 61, 69, 86, 88, 89, 94, 97, 110, 123, 126, 127, 134, 149, 151, 152, 162, 164, 169, 178, 185, 191, 203, 208, 209, 213, 216, 220, 222, 224, 230, 238, 240, 246, 249, 256, 260, 285, 286, 289, 295, 298–300, 310, 320, 338, 343, 345, 362, 366, 380, 381, 386, 387, 396, 397, 399, 403, 404, 406]. While the astonishing number of published material can be connected to the general exponential publication trend, it also indicates how privacy is becoming more and more pressing for researchers. A possible influencing factor is the significant increase in terms of IoT devices acquired by users during the pandemic years [315], which has naturally further increased the need for privacy-preventing mechanisms.

An additional preliminary view on the characteristics of the primary studies is concerned with the programming languages employed to devise the artificial intelligence techniques. We noticed that not all the articles explicitly mentioned the programming languages used. In some cases, this information has been obtained by analyzing side information, i.e., references of third-party libraries, code snippets commented in the articles, or manual analysis of replication packages. However, in 91 cases we could not find any information to elicit the programming language adopted.

Figure 8.3 provides the results of this analysis. We identified PYTHON as the key means enabling the definition of artificial intelligence techniques: this was indeed used in 67% of the articles [4, 5, 15, 27, 30, 36, 55, 110, 134, 135, 149, 158, 160, 164, 167, 178, 184, 185, 191, 193, 199, 207, 216, 233, 248, 257, 270, 278, 279, 283, 289, 298, 314, 320, 330, 362, 380, 386, 396, 403, 404]

This result was somehow expected, as PYTHON is widely considered as the main programming language for data science and machine learn-

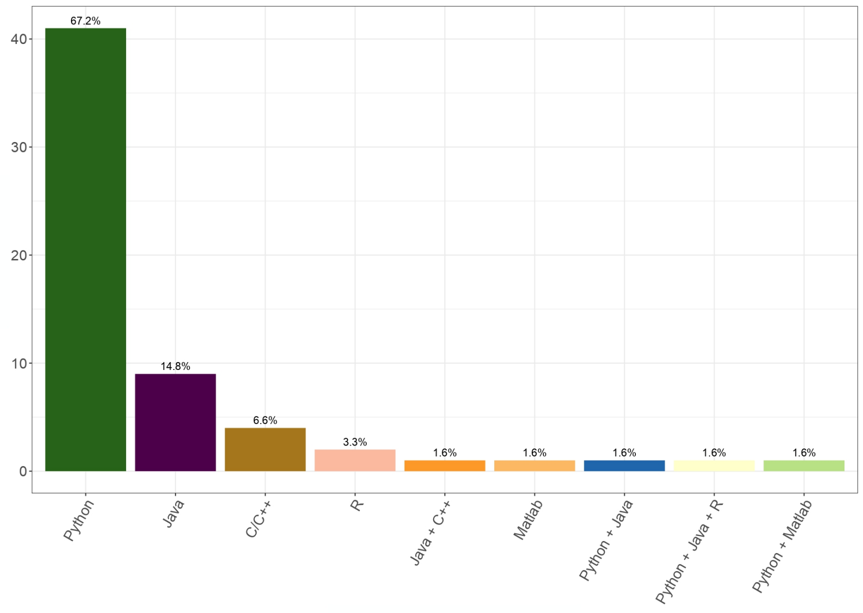

Figure 8.3: Programming languages used to devise the artificial intelligence techniques proposed in the primary studies.

ing engineering, as it offers a large amount of data science libraries that make the development of artificial intelligence techniques easier.[6] Other programming languages are less used. In 11% of the cases researchers preferred a combination of multiple programming languages. In these cases, different programming languages were used to implement different steps of the artificial intelligence pipelines: as an example, [131] employed the R toolkit to perform data cleaning operations and then relied on the *Weka* library[7]—written in JAVA—to devise a machine learning solution.

Hence, such an analysis allows us to recommend the usage of PYTHON for building novel solutions based on artificial intelligence to treat privacy concerns in IoT: this solution would indeed offer an easier chance to build

[6]Top programming languages for data science and machine learning engineering: `https://towardsdatascience.com/top-programming-languages-for-data-science-in-2020-3425d756e2a7`.

[7]The *Weka* toolkit: `https://www.cs.waikato.ac.nz/~ml/weka/`.

techniques that can extend the existing ones, e.g., by applying specific, tailored mechanisms on top of the techniques proposed in literature, or even compare the performance of the newly proposed techniques with the existing ones.

Summary.

The problem of privacy detection and preservation in IoT using artificial intelligence is now, more than ever, relevant and massively explored by researchers. The publication trend is indeed exponential and about 50% of the primary studies has been released in 2020. PYTHON is the top programming language employed to build the artificial intelligence techniques proposed in literature.

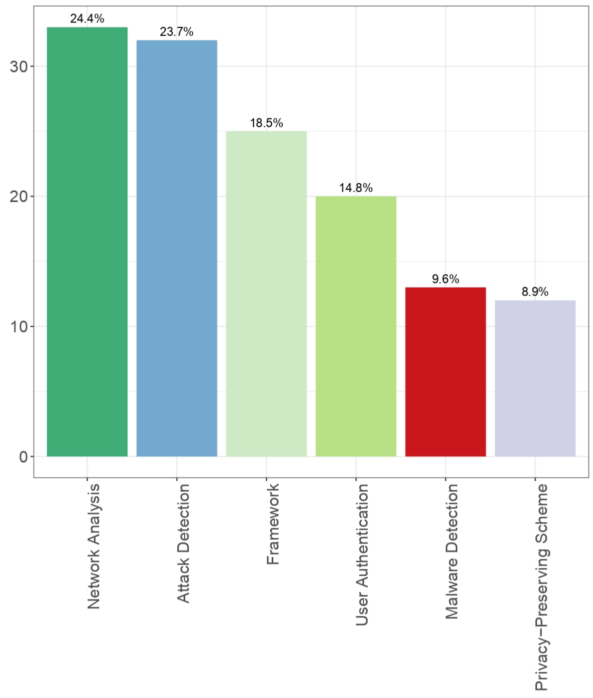

Figure 8.4: Topics frequency.

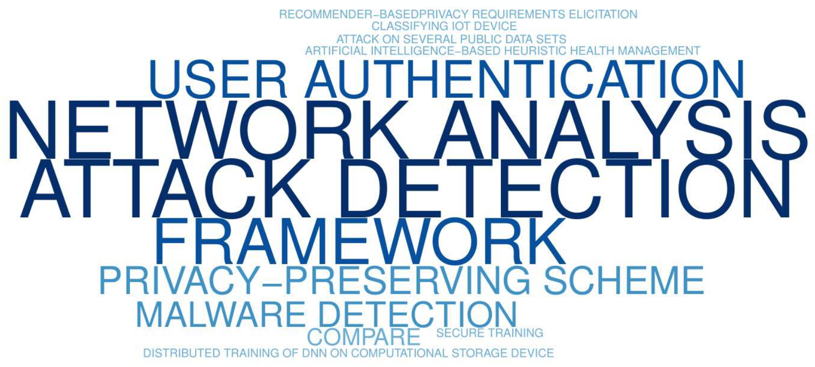

Figure 8.5: N-Gram topics treated.

RQ_1. On the privacy tasks tackled with the use of AI techniques.

To address **RQ_1**, we elicited the privacy task(s) performed in the primary studies. We labeled each paper with the set of tasks considered to enable the analysis. Figure 8.4 reports the top-6 tasks performed in the primary studies. For the sake of clarity, we focus the following discussion on these tasks since these are the ones considered by at least 10% of the primary studies. Nonetheless, we report in Figure 8.5 a word cloud that summarizes the whole set of tasks considered.

NETWORK ANALYSIS. The most prominent task is the one of *Network Analysis*—24.4% of the primary studies (33) explicitly focused on that. The authors of these primary studies highlighted a critical threat to privacy due to the fact that IoT devices typically share information without secure protocols. For this reason, data might be easily stolen [167, 307, 308, 385, 404]. More specifically, the task aims at investigating the presence of malicious traffic and/or activities on networks, e.g., the exchange of vulnerable packages. Artificial intelligence models are typically used to classify network traffic and identify sensitive or personal information transmitted by IoT devices [3, 16, 17, 43, 50, 55, 69, 88, 97, 127, 158, 160, 165, 167, 184, 196, 207, 210, 248, 256, 279, 292, 295, 307, 308, 310, 320, 354, 362, 368, 385, 404, 407]. For instance, Fei *et al.* [97] collected traffic data from the network environment to feed a *Random Forest* algorithm able to classify an abnormal traffic potentially leading to a denial of service. In a very similar fashion, the other approaches proposed in literature collect information from various sources to train machine learners able to classify malicious inputs to a network.

ATTACK DETECTION. This task has been taken into account by 23.7% of the primary studies (32). It refers to the possible detection of malicious actions. More particularly, we identified two main use cases: *"Intrusion Detection"* and *"Anomaly Detection"*. The former consists of the definition of a hardware or software component to detect possible attacks on an IoT device. The intrusion detector analyzes the network traffic and pinpoints possible suspicious activities, like phishing and ransomware. To perform this action, the authors typically used artificial intelligence to analyze this traffic, searching for anomalous patterns that can indicate an intrusion on the system [7, 25, 41, 110, 249, 283, 294, 298, 333, 338, 401]. The latter use case, i.e., *"Anomaly Detection"*, may be seen as a sub-category of the *"Intrusion Detection"* one: the purpose, indeed, is exactly the same but with a fundamental difference due to the methodology applied to identify malicious actions. While the *"Intrusion Detection"* analyzes the signatures of known attacks or possible deviations from normal traffic, *"Anomaly Detection"* relies on statistical models to verify the incoming or outgoing traffic [26, 27, 93, 131, 135, 169, 215, 263, 329, 379]. It is worth noting that, in some cases, the authors of the primary studies did not explicitly indicate the specific use case considered, i.e., they simply refer to *"Attack Detection"* (e.g., [1, 5, 15, 126, 154, 178, 185, 246, 285, 397, 409].). For this reason, we grouped *"Intrusion Detection"* and *"Anomaly Detection"* under the *"Attack Detection"* task.

FRAMEWORK BUILDING. Building a framework to characterize privacy concerns is the focus of 25 studies [30, 54, 89, 123, 155, 189, 209, 214, 216, 220, 223, 230, 238, 240, 270, 286, 289, 299, 314, 366, 367, 381, 387, 389, 405]. A typical use case is the creation of frameworks that can be then used to experiment new mechanisms to train machine learning models in a distributed environment [30, 89, 123, 214, 216, 220, 238, 240, 270, 286, 289, 299, 366, 381, 387, 405]. The framework building consists of the design and implementation of usable tools or pipeline that combine multiple artificial intelligence algorithms to detect privacy issues or preserve privacy. As an example, Meurish *et al.* [238] devised a decentralized and privacy-by-design platform that loads confidential artificial intelligence models into a trusted execution environment, hence protecting users from possible privacy concerns. On a similar note, Wang *et al.* [366] defined a federated machine learning approach that enables users to deploy complex clustering problems into the cloud.

USER AUTHENTICATION. 20 primary studies defined new secure authentication mechanisms [40, 52, 61, 62, 92, 96, 99, 111, 124, 132, 143, 151, 190, 199, 222, 224, 233, 311, 325, 396]. As an example, this category refers to the definition of person authentication tools that exploit biometric sensors: this is especially true in the healthcare field, where biometric sensors are used to monitor patients through the measurement of the blood pressure, heart rate, and others; afterward, the collected parameters are used to generate a unique identifier that can be used to access a system or in a reserved area [40, 52, 96].

MALWARE DETECTION. This task was the subject of 13 primary studies and refers to the creation of agents that analyze the processes that execute on a host machine to identify possible malware. The most common task consisted of the identification and/or classification [4, 6, 22, 35, 79, 86, 130, 162, 191, 193, 278, 305, 343] of the various types of malware. More particularly, we recognized two different trends. First, the use of pattern mining to detect malicious applications. For instance, Darabian *et al.* [79] used sequential pattern mining to detect the most frequent opcode sequences of malicious IoT applications; these sequences were then used to distinguish malicious from benign IoT applications. Second, the use of supervised machine learning approaches to classify malware. As an example [6] trained a *Random Forest* algorithm with malware data of ANDROID applications in order to identify malicious mobile apps.

PRIVACY-PRESERVING SCHEME. This task was subject of 12 primary studies and refers to the definition of new protocols and schemes to improve privacy. The authors of the primary studies typically include blockchain or similar mechanisms to keep data safe [142, 160, 164, 208, 213, 241, 260, 300, 384, 386, 399, 402]. An example is represented by the work of Zhao *et al.* [402], who devised a blockchain-based federated learning approach for IoT devices, where the data collected from multiple sensors are stored within a privacy-preserving blockchain before being consumed by machine learning models.

Other tasks are much less considered, perhaps because they represent emerging topics or because there are few datasets that may be used to perform them. We further analyze this in the context of the next RQs.

Key findings of RQ$_1$.

The results of **RQ$_1$** indicate six tasks that are often considered for the application of artificial intelligence techniques: (1) *"Network Analysis"*; (2) *"Attack Detection"*; (3) *"Framework Building"*; (4) *"User Authentication"*; (5) *"Malware Detection"*, and (6) *"Privacy-Preserving Scheme"*. A common approach is that of using artificial intelligence techniques on networks in order to detect possible reserved information exchanged or even the vulnerable IoT devices in a certain environment.

RQ$_2$. On the IoT domains where AI techniques have been applied.

When addressing **RQ$_2$**, we needed to elicit the domain from each of the considered primary studies. In this respect, we labeled each paper with one or more domains: in cases where the domain was not explicitly reported by the authors, we used the label *"Smart Environment"*: this indicates that a certain approach is generic enough to be used in more domains. The results of this analysis are depicted in Figure 8.6. As reported, the vast

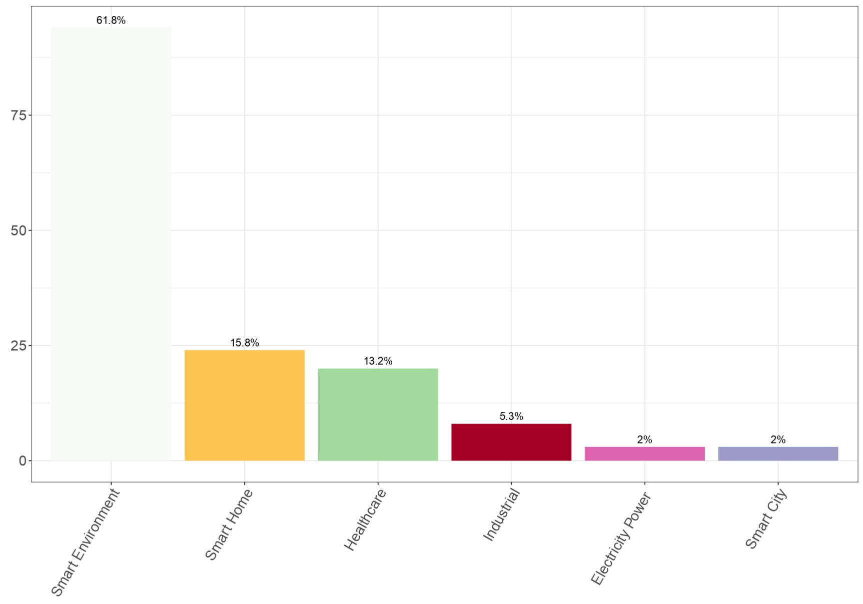

Figure 8.6: IoT domains where the artificial intelligence methods have been experimented.

majority of the primary studies do not explicitly indicate a use case domain for the artificial intelligence approach proposed or experimented. This was the case for 93 papers (61.8%). In most of these studies, the authors limit themselves to generic discussions of IoT environments where their approach might work. This indicates that most techniques are agnostic and can be applied for a variety of purposes. Typically, these have to do with domains like smart factories, military fields, smart home, healthcare, and more [7, 26, 43, 61, 134, 199, 238, 278].

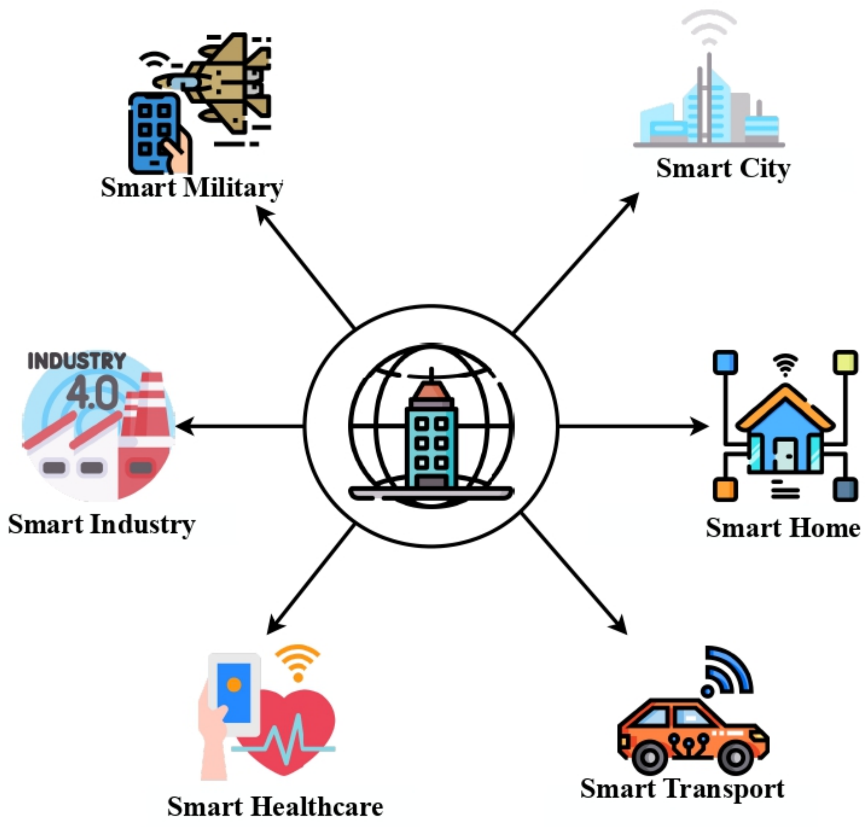

Figure 8.7: Definition of *"Smart Environment"*, according to the analysis of the papers that do not explicitly report the domains.

Figure 8.7 reports a taxonomy of the *"Smart Environment"* domain, which was built after analyzing the papers that attempted to devise agnostic techniques. Besides the generic smart environment domain, 24 studies (15,8%) discussed techniques for smart home, while other 20 (13.2%) proposed approaches to manage healthcare-related issues. There are two likely reasons behind this result. On the one hand, the research interest in smart home might be driven by the increase of IoT devices that can be used in such domain. As an example, devices like AMAZON ALEXA or GOOGLE HOME are becoming affordable and popular. As a consequence, the privacy of smart home devices represents a critical challenges to face.

On the other hand, the healthcare domain has often caught the attention of researchers, given that IoT techniques can be used to monitor patients and exchange personal data to speed up diagnoses and communications.

Table 8.3: Frequency of artificial intelligence techniques used in literature to deal with privacy concerns in IoT systems.

Artificial Intelligence Technique	Smart Environment	Smart Home	Healthcare	Industrial	Smart Cities	Sum
SVM	21	10	6	1	2	50
Random Forest	17	10	5	2	1	38
K-Nearest Neighbours (k-NN)	10	7	8	2		27
Decision Tree	8	8	5			23
Convolutional Neural Network (CNN)	11	2	4	1		19
Naive Bayes	6	4	5	1	1	18
Multilayer Perceptron (MLP)	9	3	2			14
Logistic Regression	8	2	1			13
Neural Network	6	1	3			13
K-Means	3	2	2	1		7

Other domains are, instead, less considered so far and represent emerging topics. The application of artificial intelligence to smart industry, for energy considerations or industry [16, 26, 30, 35, 54, 92, 97, 155, 178, 185, 207, 210, 213, 224, 299, 308, 397, 399, 406] was indeed the object of recent papers published in 2020. This suggests that the research community is trying to approach domains that were not typically targeted.

Key findings of RQ$_2$.

So far, most of the proposed techniques target multiple smart environments and were designed to be generic enough to work in various domains. At the same time, smart home and healthcare are established contexts where privacy concerns are always challenging. Our literature review also identified some emerging domains for artificial intelligence, like, for instance, the application of smart techniques for electricity power reduction.

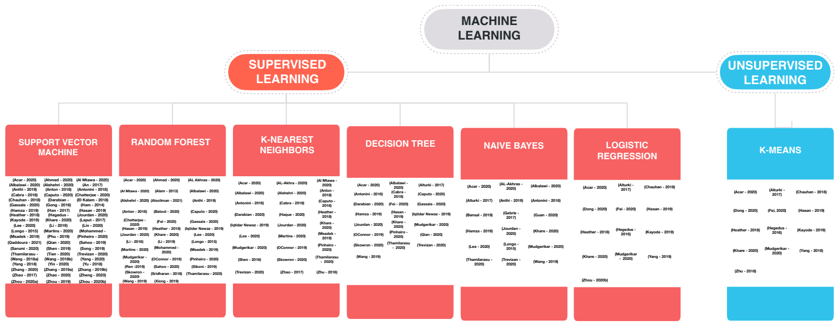

Figure 8.8: Taxonomy of the machine learning techniques used in literature to deal with privacy concerns.

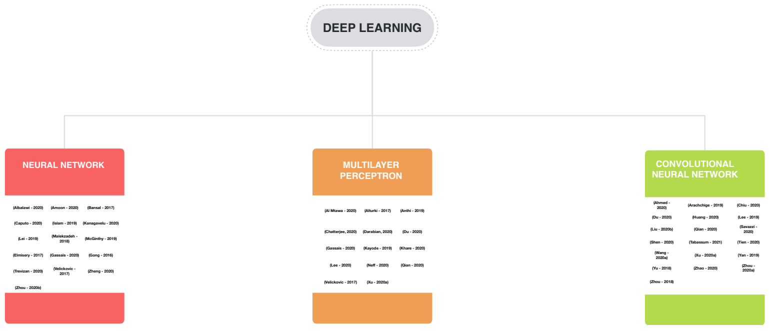

Figure 8.9: Taxonomy of the deep learning techniques used in literature to deal with privacy concerns.

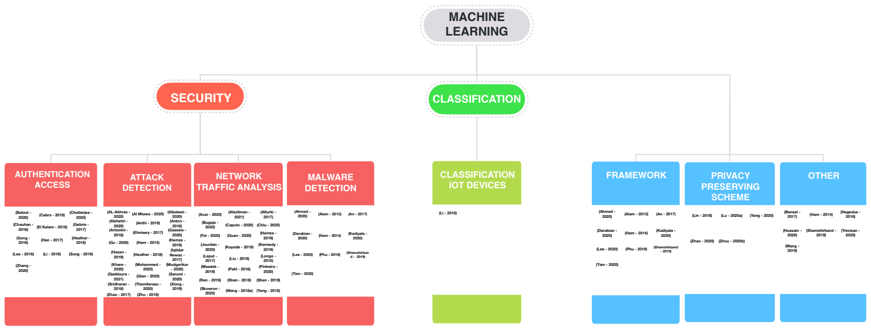

Figure 8.10: Use cases where the supervised and unsupervised learning techniques have been used.

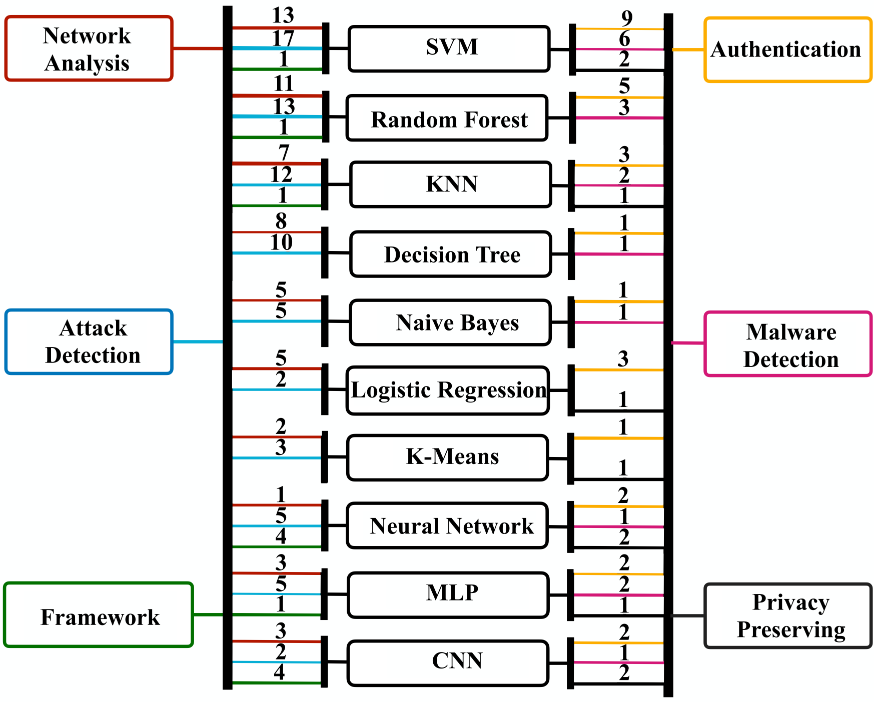

Figure 8.11: Frequencies of artificial intelligence tasks with the most six tasks considered.

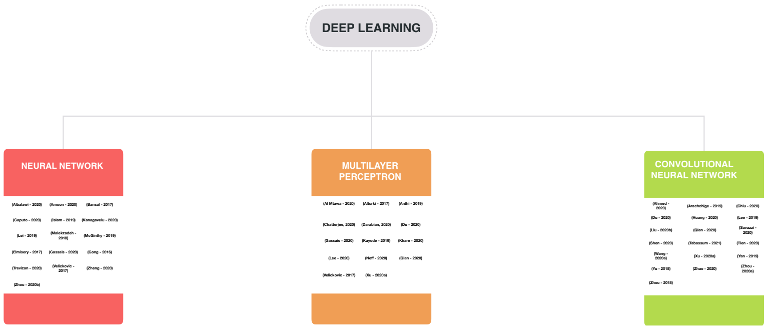

Figure 8.12: Use cases where the deep learning techniques have been used.

RQ_3. On the families of artificial intelligence algorithms used to deal with privacy in IoT systems.

With **RQ_3** we analyzed the primary studies in order to identify the artificial intelligence techniques that were used and label them according to their characteristics. Figures 8.8 and 8.9 show the results of our analysis, while Table 8.3 overviews the number of primary studies that adopted each technique, also indicating the domain where these have been experimented.

Shallow machine learning approaches, i.e., approaches that learn from data described by predefined features [382], are more frequently devised. Supervised techniques have been used in various forms: these are connected to the definition of prediction models that can distinguish the characteristics of an unseen instance based on a training base. According to our analysis, a number of algorithms have been proposed, like *Random Forest* [1, 3–7, 15, 16, 25, 26, 40, 55, 61, 97, 110, 135, 143, 154, 160, 169, 191, 196, 199, 210, 230, 246, 248, 249, 257, 279, 292, 295, 314, 320, 329, 369, 379], *Support Vector Machines* [3–5, 7, 15, 22, 25–27, 52, 55, 61, 62, 79, 93, 110, 124, 130–132, 135, 143, 144, 160, 165, 169, 184, 191, 199, 203, 210, 230, 246, 248, 278, 279, 283, 285, 295, 298, 308, 325, 338, 343, 345, 368, 369, 385–387, 389, 396, 397, 399, 401–406]. None of the surveyed papers provided motivations leading to the selection of these algorithms. Nonetheless, the higher amount of primary studies proposing supervised learning techniques is likely due to the characteristics of the problems considered: as a matter of fact, most researchers have been working on the definition of classification and/or regression approaches to identify privacy concerns, which calls for the adoption of supervised machine learning techniques. An overview of the privacy tasks considered with each of the machine learning techniques is provided in Figure 8.10. As shown, typical use cases are the authentication problem and the network traffic analysis. The authors that considered this authentication problem typically applied supervised learning algorithms to classify authorized or unauthorized accesses. Network analysis was instead approached by collecting previous network data and features in order to devise prediction models that could discriminate the likelihood that the current traffic is anomalous and may therefore lead to security threats for an IoT device.

A lower amount of studies focused on unsupervised learning. Precisely, the use of clustering, and the *k-Means* algorithm in particular, allowed researchers to devise approaches able to group together the common properties that may characterize the privacy concerns treated. Clustering algorithms were used to cover two macro-areas: data classification and devices aggregation. The former refers to clustering to classify devices or to perform network traffic tasks. The latter refers to the definition of common features or parameters related to IoT devices [7].

During our analysis, we found that the clustering algorithms were either used as an alternative to supervised learning algorithms [7] (e.g., to classify or aggregate devices based on some criterion for instance defined commons features or parameters related to IoT devices) or in combination with them [26, 127, 131, 132, 233, 238, 366, 369, 406]). As an example, the studies performed by Anton et al. [26] and Hamza et al. [131] employed clustering to classify abnormal network traffic, hence defining unsupervised approaches that could identify possible anomalies on a network. At the same time, an example of combination was presented in the paper by Hag et al. [132], who focused on the problem of occupancy detection, i.e., the classification of whether a room is occupied by a person. In this case the authors used time-stamped images of environmental variables like temperature, humidity, light, CO2, to assess the accuracy of a user authentication approach. When gathering the images, the authors applied a k-means clustering algorithm to define a first grouping of normal and malicious room occupancy. These clusters were used to obtain labels that were later exploited to train an *SVM* algorithm.

A more recent trend is the adoption of *deep* learning. We observed that the primary studies that used this type of learning were all published in 2020, indicating a growing interest.

Figure 8.11 provides a conclusive overview on the artificial intelligence techniques used in literature. In particular, the figure connects the top-6 tasks coming from the results of **RQ**$_1$ to the artificial intelligence techniques adopted to solve them. Each task is depicted with a different color; this color characterizes the edges that connect each task to the techniques used in literature. The edges are weighted based on the amount of primary studies using a technique to address a certain task. For instance, the *"Network Analysis"* task is reported in red. The red edges indicate that the task has been addressed in 13 papers with the use of *SVM*, in 11 papers with *Random Forest*, in 7 papers with *KNN*, and so on. From the figure, we can confirm that *Support Vector Machine* has been the most used artificial intelligence algorithm (48 times) to address the majority of the privacy tasks investigated by researchers so far.

Figure 8.12 overviews the tasks faced by researchers through the use of deep learning. To provide an example of a common task for which deep learning has been used, let consider the OCCLUMENCY framework developed by Lee et al. [189]. This is a cloud-driven solution designed to protect user privacy without reducing the benefits of cloud resources. It is common for IoT applications to collect and share sensitive information through a cloud platform. The OCCLUMENCY framework uses deep learning to encrypt that information without increasing the latency of the cloud platform's response. More in general, we noticed that deep learning had been experimented for tasks previously treated with shallow machine learning techniques also to verify how deep learning approaches can improve the prediction performance of traditional shallow learning algorithms.

As an outcome of our analysis, there are two main observations to make. First, most researches focused on the adoption of supervised learning, while other types of artificial intelligence techniques seem to have been neglected. As such, our systematic literature review suggests that additional analyses might focus on unsupervised learning and orthogonal techniques, like evolutionary algorithms or pattern recognition. Secondly, we identified only three empirical investigations aimed at comparing the various forms of artificial intelligence techniques employed [36, 369, 403]. In this sense, we highlight further possibilities for the empirical software engineering community, which might exploit our literature survey's outcome to design and execute empirical investigations into the matter.

Key findings of RQ$_3$.

The results obtained from **RQ**$_3$ indicate that the large majority of primary studies focused on supervised learning techniques to deal with privacy concerns. Yet, we highlight the lack of analyses on other types of artificial intelligence approaches, other than the lack of empirical studies to compare the existing techniques.

Table 8.4: List of datasets used to device or experiment the artificial intelligence techniques proposed in literature.

Dataset	Category	Task	Paper	Link
MNIST	Handwritten	Comparison	[29, 144, 384]	yann.lecun.com/exdb/mnist/
		Network Analysis	[158, 404, 407]	
		Framework	[89, 209, 214, 216, 240, 270, 286, 309, 367, 381]	
		Privacy Preserving Scheme	[208, 380, 399, 402]	
		User Authentication	[396]	
CIFAR-10	Image Classification & Object Detection	Comparison	[29, 384]	www.cs.toronto.edu/~kriz/cifar.html
		Network Analysis	[69, 407]	
		Framework	[240, 270, 286, 367, 381]	
		Privacy Preseving Scheme	[208]	
KDD Cup 99	Cybersecurity	Attack Detection	[215, 329, 333, 401]	www.kdd.org/kdd-cup/view/kdd-cup-1999/Data
DS2OS	Cybersecurity	Attack Detection	[135, 169, 185]	www.kaggle.com/francoisxa/ds2ostraffictraces
Adult	Personal Information	Comparison	[403]	www.kaggle.com/wenruliu/adult-income-dataset
		Network Analysis	[127]	
Heart Disease	Healthcare	Secure Training	[134]	archive.ics.uci.edu/ml/datasets/heart+disease
		Network Analysis	[308]	
CASIA-WebFace	Face Recognition	Network Analysis	[368]	paperswithcode.com/dataset/casia-webface
		Framework	[309]	
CTU-13	Cybersecurity	Comparison	[36]	www.stratosphereips.org/datasets-ctu13
		Malware Detection	[305]	
Fashion MNIST	Object Detection & Image Classification	Framework	[214, 270]	www.kaggle.com/zalando-research/fashionmnist
GeoLife	Tracking GPS	Attack Detection	[379]	www.microsoft.com/en-us/download/details.aspx?id=52367
		Framework	[238]	

RQ$_4$. On the datasets employed by the artificial intelligence methods.

After providing an overview of the tasks, domains, and families of techniques employed to deal with privacy concerns in IoT systems, we started our fine-grained analysis of the design and evaluation of these techniques. With **RQ**$_4$, we collected data and characteristics of the datasets used by the primary studies.

Table 8.4 reports the list of datasets, along with information on their category, the tasks for which they were employed, and where to find them.

Figure 8.13 shows top-10 frequency of use of each dataset.

We observed that most of the primary studies (42%, 64 papers) relied on *"MINIST"*, while the others have been exploited to a lower extent. More specifically, let us comment on those datasets:

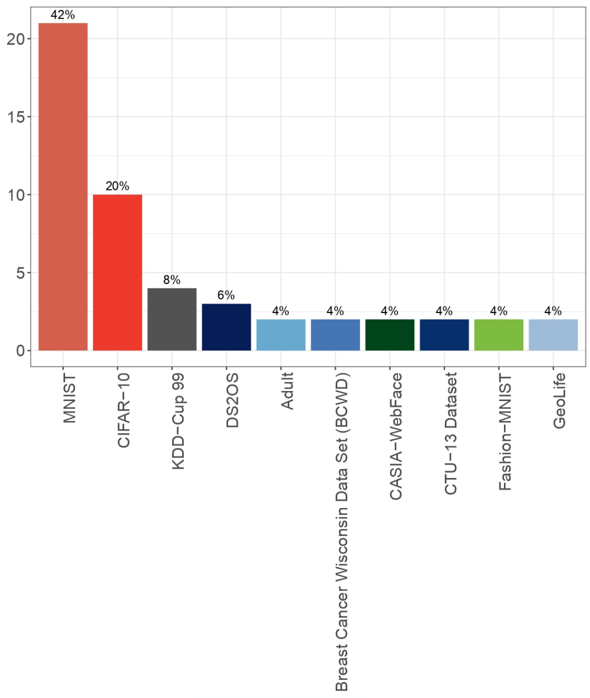

Figure 8.13: Frequency of use of the datasets exploited by the primary studies.

MNIST. This is a dataset of handwritten digits created for the specific purpose of experimenting ML techniques. It contains about 60,000 examples and a test set of 10,000 samples. It was used in 42% of the papers [29, 89, 144, 158, 208, 209, 214, 216, 240, 270, 286, 367, 380, 381, 384, 396, 399, 402, 404, 407], and is particularly indicated to test authentication techniques that rely on biometric data. For instance, Jiang *et al.* [158] experimented with biometric data obfuscation techniques to verify how the digits classification performance of a deep neural network vary with respect to the case where the network is trained with the original, cleaned digits.

CIFAR-10. It consists of 60,000 images categorized in 10 classes. The dataset has been created for the specific case of machine learning, as it contains around 6,000 images for each class and is released so that a researcher can use 50,000 images for training and 10,000 images for testing machine learning techniques. We have found ten papers that used this dataset [29, 69, 208, 240, 270, 286, 367, 381, 384, 407]. Typically, it is used to experiment classification algorithms aiming at addressing the privacy concerns of images and videos, like the problem of understanding whether sensitive data can be derived from fragments of images captured by sensors [240, 381, 407].

KDD CUP 99. This dataset contains raw signals obtained in nine weeks from the *TCP dump*. The dataset includes 24 training attacks and 14 types of test data. This dataset is used for the *intrusion detection* learning task and to build supervised algorithms that could learn from these examples to predict the emergence of intrusion attacks [215, 329, 333, 401].

DS2OS. The dataset contains traffic traces obtained in IoT environments and is typically used to verify *anomaly detection* algorithms [135, 169, 185]. For instance, Hasan et al. [135] employed anomaly detection on this dataset to identify possible attacks in IoT sites.

ADULT DATASET. It contains information about people, including the annual income. This is typically exploited by researchers interested in building and/or assessing techniques to detect personal data losses. Similar to the cases above, the dataset seems to be particularly useful for classification algorithms, given that it reports labeled data that can be used for training purposes [127, 403].

BREAST CANCER WISCONSIN DATASET. This dataset is computed from a breast mass digitization image of fine needle aspirate (FNA). The dataset contains 569 instances and 32 attributes (including symmetry, concave points, area). Researchers have been using the dataset to experiment with supervised *classification* techniques that aim at identifying potential personal data losses [134, 308].

CASIA-WEBFACE. The dataset contains 494,414 face images of 10,575 real identities. This is typically used for face verification and face identification tasks [309, 368]. For instance, Wang et.al. [368] used this dataset to experiment with a combination of *Deep Neural Network* and *Support Vector Machines* able to analyze video streams and identify and blur faces based on privacy policies, obtaining an accuracy rate close to 92%.

CTU-13. The dataset contains botnet traffic captured in regular traffic and background traffic. Researchers used the dataset for *malware detection* tasks [36, 305]. As an example, Bansal et.al. [36] employed multiple machine learning algorithms, including *Naive Bayes* and *Neural Networks*, to detect botnets, obtaining F-1 scores up to 88%.

FASHION MNIST. This dataset includes ZALANDO's article and contains about 60,000 training set images and 10,000 test set examples. It is divided into 10 classes (including T-shirts, pullovers, and coats), and each category contains 10,000 examples [214, 270]. The studies that exploited this dataset were interested in building techniques that might prevent privacy leaks due to the identification of people from their clothes.

GEOLIFE. This dataset contains GPS trajectories collected in over three years. The dataset includes information about the time-stamped and information about the latitude, longitude, and altitude [238, 379]. It has been used to train and test techniques that could prevent the localization of people based on their coordinates.

To broaden the scope of the discussion, it is worth focusing on the tasks for which each of the above datasets has been used—Table 8.4 reports the details of our analysis. We could first notice that the *"MINIST"* dataset has been used by multiple authors to pursue several tasks: authors opted for it when performing comparisons among artificial intelligence techniques [29, 144, 384], building frameworks [89, 209, 214, 216, 240, 270, 286, 309, 367, 381] or experimenting with network analysis approaches and authentication mechanisms, other than verifying the accuracy of privacy-preserving schemas. Similar considerations and conclusions can be drawn for the *"CIFAR-10"* dataset. These observations highlight the flexibility of the datasets with respect to multiple tasks. On another note, the other datasets have been used in a more restrictive manner and mostly for dealing with the task of *"Attack Detection"*. This is even the only task considered when employing the *"DS20S"* and *"KDD Cup 99"* datasets.

While we already discussed about the availability of only a few datasets for experimenting with artificial intelligence techniques, some additional observations should be made. The authors of the *"Fashion MNIST"* dataset openly criticized the original *"MNIST"* dataset. Indeed, its structure allows both traditional and deep learning algorithms to achieve very high accuracy without providing enough insights into the actual validity of the predictions performed. In other words, the dataset is built in a biased way that impacts the analysis of the real capabilities of the experimented techniques. Other data scientists and practitioners also remarked this. In April 2017, a GOOGLE BRAIN research scientist and deep learning expert, Ian Goodfellow, advised migrating to other datasets. Later on, another deep learning expert, François Chollet, explained that the *"MNIST"* dataset is not good at representing everyday tasks. These considerations, along with the consideration that a large number of primary studies employed this dataset, allow us to claim that the research in privacy of IoT devices might require a critical re-assessment. This is further confirmed by the fact that 19 primary studies evaluated the proposed approaches only in terms of accuracy [29, 89, 144, 158, 208, 209, 214, 216, 270, 286, 309, 367, 380, 381, 384, 396, 402, 404, 407], hence possibly biasing the conclusions drawn—more details are reported when addressing **RQ**$_6$.

Another discussion point concerns with the intrinsic characteristics of the datasets. When addressing **RQ**$_4$, we analyzed whether and to what extent the available datasets are balanced. Data balancing is a crucial data quality aspect to take into account while selecting a suitable dataset to create and/or validate privacy approaches [39]. The availability of balanced datasets, namely of datasets for which there are a similar amount of data for each class, might notably affect the learning capabilities of artificial intelligence techniques [39], other than implying the definition of methodological steps that aim at facing this potential learning bias.

Table 8.5: Classify balanced or unbalanced datasets

Dataset	Balanced
MNIST	Yes
CIFAR-10	Yes
KDD Cup 99	Yes
DS2OS	No
Adult	No
Breast Cancer Wisconsin DataSet (BCWD)	No
CASIA-WebFace	No
CTU-13	No
Fashion MNIST	Yes
GeoLife	No

Table 8.5 summarizes our analysis on the data balancing of the considered datasets. The three most used datasets are balanced, while *"DS20S"* and *"Breast Cancer Wisconsin DataSet"* are not. Researchers can use this information to take appropriate data balancing considerations during the design of their studies, other than exploit it to analyze deeper the validation of the proposed techniques (**RQ**$_6$).

Key findings of RQ$_4$.

We point out the need for further open datasets that may cover a larger variety of privacy concerns. About 40% of the papers conducted experiments on the *"MNIST"* dataset (an Handwritten dataset). Nonetheless, it has been criticized, as it may lead to biased interpretations of the results obtained by artificial intelligence techniques. As a consequence, the conclusions drawn by most of the papers published so far might need to be re-assessed.

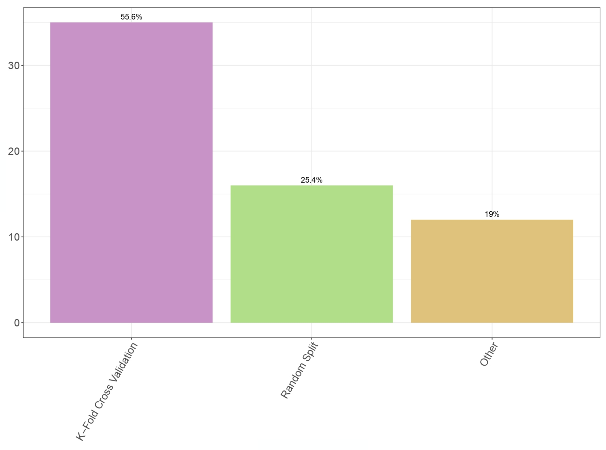

Figure 8.14: Validation techniques used by the primary studies.

RQ_5. On the validation strategies employed to assess the artificial intelligence methods.

In the context of **RQ_5** we analyzed the validation strategies adopted when assessing the capabilities of artificial intelligence methods devised to deal with privacy concerns in IoT systems. As clarified in Section 8.2, not all the primary studies validated the proposed techniques. This was the case for 21 papers [7, 50, 92, 99, 111, 142, 149, 152, 155, 207, 223, 224, 233, 240, 289, 299, 330, 366, 385, 386, 406]: hence, this research question takes the validation procedures of 131 primary studies into account. Figure 8.14 shows the results of our analysis. 55.6% of the studies (35) that explicitly indicate the validation technique used the so-called k-fold cross validation [3, 4, 6, 16, 17, 29, 35, 40, 55, 62, 110, 124, 130, 132, 135, 143, 190, 191, 199, 210, 238, 246, 248, 257, 285, 292, 295, 300, 308, 320, 329, 343, 345, 354, 369], with a value of k equals to 5 or 10. This is a method that can be used to estimate the performance of machine learning algorithms: it randomly splits a dataset into k groups called folds, and (I) takes one fold as test and k-1 folds as training, (II) fits a machine learning model and executes it on the current test fold, and (III) iterates the procedure until all unique folds have been considered exactly once as test set. Upon completion of the validation procedure, the results obtained are summarized employing statistical indicators like, for instance, the mean number of true positive instances identified over the various validation runs.

When analyzing the primary studies, we could not identify the specific reasons leading researchers to use this validation procedure. We cannot provide insights on whether its adoption was the most suitable or whether other validation procedures would better fit the specific problems treated in the studies. The only exception to this general discussion concerns the work by Meurisch et al. [238]. The study proposed an AI-based privacy-preserving mechanism to overcome the need to continuously share user data streams in the cloud, which was later validated through 10-fold cross-validation. The proposed approach employs temporal data, namely data collected over a given time frame and that, for this reason, follows a temporal order. The application of cross-validation in this scenario risks bias the interpretation of the results [8]. Indeed, the cross-validation might accidentally lead future data to be used for training past data, causing a form of data leakage that interprets results biased. Unfortunately, we could not understand if and how the authors have mitigated the risks connected to the adoption of cross-validation. Yet, we can argue that more details on the rationale and the methodology adopted to validate the artificial intelligence methods would be required to properly assess the validity of the insights provided.

Besides cross-validation, another popular strategy is the so-called random split or percentage split. This randomly splits the dataset into training and test sets, e.g., retaining 80% for training and 20% for testing [5, 69, 89, 96, 134, 151, 185, 196, 209, 223, 283, 305, 310, 333, 380, 386, 403]. Similarly to the discussion above, the primary studies that employed this validation did not explicitly mention the rationale behind its use nor the possible threats that this validation might cause.

The last 29 primary studies used different strategies, which we grouped as *"Other"* in Figure 8.14. These studies employed various validation methods, e. g., the Monte Carlo cross-validation [230].

[8]https://medium.com/@soumyachess1496/cross-validation-in-time-series-566ae4981ce4

Table 8.6: Datasets used and the training/validation strategies employed.

Dataset	Training/Validation Strategy
MNIST	K-Fold Cross-Validation, Random Split
CIFAR-10	K-Fold Cross-Validation, Random Split
KDD Cup 99	K-Fold Cross-Validation
DS2OS	K-Fold Cross Validation, Random Split
Adult	Random Split
Breast Cancer Wisconsin DataSet (BCWD)	K-Fold Cross-Validation
CASIA-WebFace	Other
CTU-13	Other, Random Split
Fashion MNIST	Not specified
GeoLife	K-Fold Cross Validation

Key findings of RQ$_5$.

Cross-validation and random split are the two most common validation procedures in the literature. However, the methodological choices behind selecting these strategies are often unclear or unspecified. This can lead to biased interpretations of the performance of artificial intelligence methods. Therefore, we argue for more detailed reporting to enhance the rigour, reproducibility, and replicability of research.

RQ$_6$. On the evaluation metrics employed to assess the AI methods.

The last perspective of our study was concerned with the evaluation metrics employed to measure the performance of the artificial intelligence techniques proposed in the literature.

The results for **RQ$_6$** are plotted in Figure 8.15. We found that a large quantity of papers only relied on the accuracy metric[5, 6, 16, 17, 29, 30, 40, 41, 55, 61, 69, 86, 88, 89, 96, 123, 127, 132, 144, 158, 160, 167, 178, 184, 189, 190, 196, 203, 208–210, 214, 216, 238, 246, 257, 260, 270, 286, 294, 300, 308, 309, 311, 325, 338, 354, 362, 367, 368, 380, 381, 384, 387, 396, 402–404, 407], that is, the total amount of correct predictions with respect to all the predictions

Table 8.7: Evaluation metrics used when working on unbalanced datasets.

Paper	Dataset	Evalation Metrics
Attack and anomaly detection in IoT sensors in IoT sites using machine learning approaches	DS20S	Accuracy, Precision, Recall, F1-Score, Roc Curves
A Novel Attack Detection Scheme for the Industrial Internet of Things Using a Lightweight Random Neural Network	DS20S	Accuracy, Precision, Recall, F1-score
Ensemble Learning for Detecting Attacks and Anomalies in IoT Smart Home	DS20S	Accuracy, Precision, Recall, F1-Score
Preserving User Privacy for Machine Learning: Local Differential Privacy or Federated Machine Learning?	Adult	Accuracy
A Differentially Private Big Data Nonparametric Bayesian Clustering Algorithm in Smart Grid	Adult	Accuracy
Privacy-Preserving Support Vector Machine Training Over Blockchain-Based Encrypted IoT Data in Smart Cities	Breast Cancer Wisconsin DataSet (BCWD)	Accuracy
Privacy-preserving k-nearest neighbors training over blockchain-based encrypted health data	Breast Cancer Wisconsin DataSet (BCWD)	Accuracy, Precision, Recall

output. In other cases, the primary studies used a combined approach, for instance by computing accuracy and precision, recall and precision, and so on. In any case, there are two observations to make on the choice of the evaluation metrics. In the first place, and similarly to the discussion done in **RQ**$_5$, most of the surveyed studies did not report on the rationale for using these metrics nor on their suitability for the considered problem. As an example, let consider the case of accuracy. By definition, the value of the metric increases as the number of both true positives and negatives increases. Some of the datasets currently available are strongly unbalanced and contain only a few elements characterizing privacy issues—this is, for example, the case of the *"Adult"* and *"Breast Cancer Wisconsin DataSet"* discussed in **RQ**$_4$. For these datasets, one is reasonably interested in assessing the performance of artificial intelligence techniques with respect to their capabilities in correctly predicting the privacy issues appearing in the minority class of the dataset. Nonetheless, training an AI-based solution with only a few instances of the class of interest might lead the approach not to properly learn how to classify them. On the contrary, the approach might be biased toward the classification of the majority class,

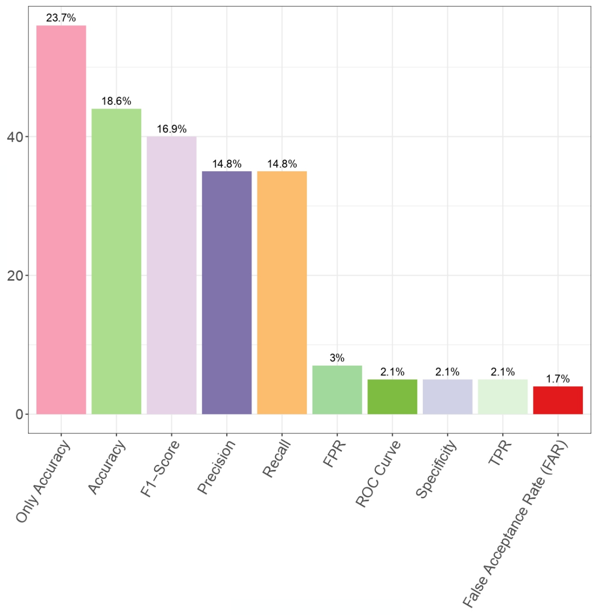

Figure 8.15: Evaluation Metrics Techniques.

namely the set of instances that do not present any privacy concern. As a consequence, measuring the accuracy metric might provide a wrong view of the performance, since the metric tends to reward the approach independently from which class it is able to correctly predict. High accuracy values can therefore indicate that the artificial intelligence approach is able to correctly predict the majority class, which is the least interesting. We could not find considerations of this type in the primary studies considered and, unfortunately, this might have had an impact on the conclusions drawn by various studies. More specifically, Table 8.7 reports the primary studies that have worked on unbalanced datasets; the last column of the table also reports the evaluation metrics considered. As it is possible to observe, all of them relied on the accuracy—for some of them, this was the only metric considered—but, perhaps more importantly, only a few assessed the performance of the techniques in a more comprehensive manner through other metrics. A second point of discussion is still related to the relation between the datasets exploited and the metrics used for evaluation. Even the primary studies that worked on balanced datasets typically relied on the accuracy. While in these cases the use of accuracy was more reasonable, some peculiarities of the datasets might have influenced the choice of the evaluation metric. For instance, as discussed in **RQ**$_4$, most of the primary studies relied on the *"MNIST"* dataset, which turned to be somehow biased toward accuracy, i.e., as already explained, the major criticism made was concerned to the fact that any AI-based technique can easily reach high accuracy levels on this dataset. As such, it is likely to believe that a re-evaluation of the techniques proposed in literature might be beneficial for the research community in order to more appropriately understand the actual value of those techniques.

Key findings of RQ$_6$.

23% of the primary studies only used accuracy to evaluate the quality of the artificial intelligence techniques. However, the characteristics of the datasets might make them biased toward accuracy, implying a biased interpretation of the real capabilities of the proposed techniques.

8.4 THREATS TO VALIDITY

As any other SLR, ours has some limitations that may have threatened the validity of the reported findings. This section discusses them along with the mitigation strategies employed to address them.

LITERATURE SELECTION. A critical challenge for a systematic literature review consists of identifying a complete set of papers to enable a comprehensive overview of the state-of-the-art. In this respect, we have first defined a search query having the goal of retrieving as many papers related to the use of artificial intelligence for dealing with privacy of IoT systems as possible without any temporal limitations. This choice has implied a higher effort in terms of manual analyses, we preferred it for the sake of completeness. Furthermore, we identified synonyms or alternative spellings of terms typically used in literature when defining the search query. In addition, we checked the presence of the search terms within existing systematic literature reviews on IoT privacy to find possible additional terms. To further increase the completeness of our study, we also conducted a backward snowballing session on the papers that passed the exclusion/inclusion criteria. Combining these actions makes us confident of the completeness of the literature selection. Nevertheless, for the sake of verifiability and replicability, we have provided as additional contribution an online appendix reporting all steps and intermediate results of our analyses [371].

LITERATURE ANALYSIS AND SYNTHESIS. Upon completion of the selection process, we applied specific exclusion criteria intending to filter out papers that could not contribute or could provide a limited contribution toward the summarization of state-of-the-art related to the defined research questions. Moreover, we did not limit the selection of primary studies to those that successfully passed the inclusion criteria, but also conducted an additional quality assessment to verify their actual suitability to our purposes. Such a manual assessment has further limited the risk of including resources that did not fit our purposes.

9 A CONCEPTUALIZATION AND ANALYSIS ON AUTOMATED STATIC ANALYSIS TOOLS FOR VULNERABILITY DETECTION IN ANDROID APPS

9.1 INTRODUCTION

The last decades and, most notably, the recent years have seen a drastic change in the way people communicate and interact among them. Around 80% of the global population owns a smartphone [150] and about 70% of these smartphones rely on the ANDROID operating system [243]. The diffusion of this operating system (OS) is favored by multiple factors, including the availability and marketing of mobile apps through the online app store [227]. In this context, previous research has pointed out that ANDROID apps can be affected by severe vulnerabilities that can impact both user privacy and security [100, 250, 365]. For this reason, several automated static analysis tools have been proposed to detect security concerns and assist mobile developers in improving their applications [179]. Nevertheless, in our research, we observed a lack of knowledge about the real support provided by these tools. In particular, it is unclear the set of problems that these tools can detect and how they behave when detecting vulnerabilities, e.g., whether their analysis fails in certain cases, the most common vulnerabilities identified, and to what extent different tools cover different vulnerabilities. An improved understanding of these aspects is crucial to let developers be aware of what kind of problems can be currently detected, other than letting them (I) more wisely select the tools to employ, (II) evaluate on complementing more tools, or (III) even understand whether current tools can actually identify vulnerabilities that are becoming more and more popular and harmful nowadays.

We focus on three automated static analysis tools, i. e., ANDROBUGS2,[1] TRUESEEING,[2] and INSIDER,[3] to elicit a taxonomy of security-related concerns detectable with these tools. Afterward, we execute the tools on 6,500 free apps to assess the number of vulnerabilities the tools can detect and the complementarity among them.

The main results of the study indicate that in most cases the tools can detect the same vulnerabilities but using a different vocabulary, causing possible misunderstandings. The tools are also complementary, which implies that developers should select tools based on the specific categories of vulnerabilities they would detect. Lastly, the considered tools only partially cover the most widespread vulnerabilities classified by the Open Web Application Security Project (OWASP) Foundation.

To sum up, we provide the following contributions:

1. An empirical investigation into the support provided by three state-of-the-practice tools for the detection of security-related concerns;

2. An empirical analysis of the complementarity among the three considered tools, which might open new research directions connected to their combination;

3. A publicly available replication package [259], which contains data and scripts employed to address and extend our research questions.

9.2 RESEARCH QUESTIONS AND METHOD

The *goal* of the empirical study was to assess the current support provided by existing automated static analysis tools in terms of vulnerability detection in ANDROID applications, with the *aim* of providing initial insights on the capabilities of these tools in terms of security issue types detected and their detection capabilities. The *perspective* is of both researchers and

[1]https://github.com/androbugs2/androbugs2
[2]https://github.com/alterakey/trueseeing
[3]https://github.com/insidersec/insider

practitioners: the former is interested in the capabilities of existing tools to evaluate whether and which aspects should be further improved; the latter are interested in understanding how existing tools can support them in their daily tasks. We set out two main research questions.

First, we analyzed the types of vulnerabilities that current tools can detect through static analysis. Our goal was to elicit a *taxonomy* of the security issue types whose identification is supported by the existing instruments. An improved understanding and investigation of this research angle are required to let researchers be aware of where to invest future research efforts, other than to let practitioners know which tools can be used to detect specific vulnerabilities, hence easing the selection of the proper tools to use in their contexts. This reasoning led to the definition of our first research question (**RQ**$_1$):

🔍 **RQ**$_1$**.** *What are the vulnerability types identified by existing automated static analysis tools for mobile apps?*

Once we had identified the types of security-related issues whose detection is supported by existing tools, we sought to provide insights into their detection capabilities. The aim is to elaborate on the extent to which existing tools can detect vulnerabilities in the first place and, if so, with which frequency they can detect the vulnerabilities types identified in **RQ**$_1$ and which complementarity exists among them.

This perspective is key to understanding the extent to which different tools can collect and provide information on different security-related concerns. From the practitioner's perspective, this analysis would ease further the tool selection process, which might be done by considering the capabilities of the tools. Also, researchers may exploit our findings to assess where additional improvements are needed, e.g., by understanding the characteristics of the tools that should be improved. Therefore we asked our second research question (**RQ**$_2$):

RQ$_1$. *What are the capabilities of existing automated static analysis tools in terms of mobile app analyzability, frequency of detection, and complementarity among them?*

The expected outcome of our analysis is an initial conceptualization and analysis of the current state of the art in vulnerability detection in ANDROID applications. When conducting the empirical study, we followed the empirical software engineering principles and guidelines described by Wohlin et al. [376]. Additionally, in terms of reporting, we employed the *ACM/SIGSOFT Empirical Standards*.[4] For the sake of replicability and reproducibility, we made available in the online appendix [259] datasets, scripts, and the additional analysis that address our research questions.

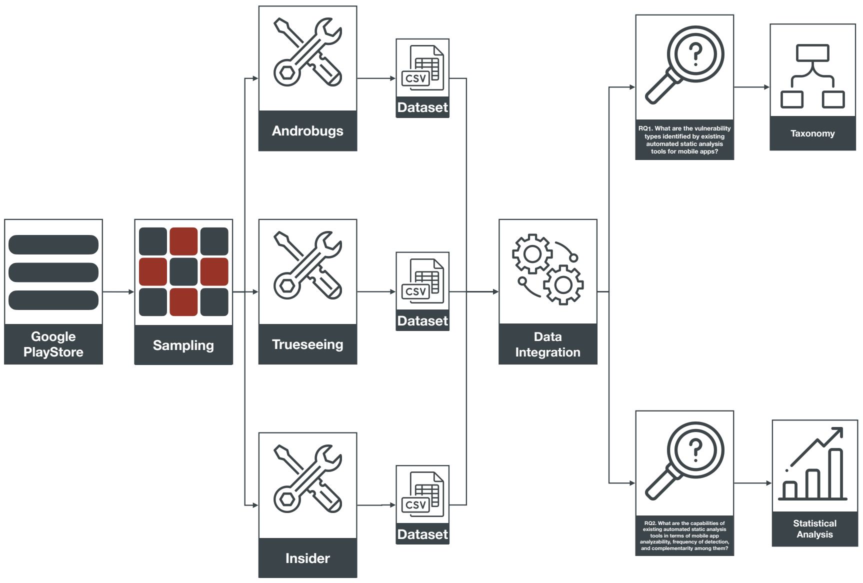
Androbugs
CSV
Dataset
Google PlayStore
Sampling
Trueseeing
CSV
Dataset
Insider
CSV
Dataset
Data Integration
RQ1. What are the vulnerability types identified by existing automated static analysis tools for mobile apps?
Taxonomy
RQ2. What are the capabilities of existing automated static analysis tools in terms of mobile app analyzability, frequency of detection, and complementarity among them?
Statistical Analysis

Figure 9.1: Overview of the Research Method.

Figure 9.1 provides an overview of the research method applied in this study. The additional details will be explained in the next sections.

[4]Available at: https://github.com/acmsigsoft/EmpiricalStandards. Given the nature of our study and the currently available empirical standards, we followed the *"General Standard"* and *"Repository Mining"* guidelines.

Context Selection

The *context* of the empirical study was composed of automated static analysis tools (**RQ**$_1$) and mobile applications (**RQ**$_2$).

To select the tools, we adopted four criteria: They were tools (I) open-source and available on GITHUB; (II) that take an `apk` file as input; (III) that perform a static analysis of the source code; and (IV) that can be run using the command line. These filters led us to consider:

ANDROBUGS2. This tool can detect 52 categories of security-related concerns, such as permission issues and exposure of sensitive information.

TRUESEEING. This analyzer can detect 7 types of security issues: *Improper Platform Usage*, *Insecure Data*, *Insecure Communications*, *Insufficient Cryptography*, *Client Code Quality Issues*, *Code Tampering* and *Reverse Engineering*.

INSIDER. According to the official documentation,[5] the tool covers the OWASP mobile Top 10 vulnerabilities and supports multiple programming languages like JAVA, KOTLIN, SWIFT, .NET and others.

When it comes to mobile apps, we aimed at analyzing a large and representative sample of mobile applications, which might let us provide sound and reliable conclusions. While researchers have released various mobile app datasets over the last decade [82, 113, 198], most of them are outdated, contain toy apps, or apps that no longer exist [112]. Therefore, we relied on a public dataset available on KAGGLE,[6] namely *GooglePlayStore dataset*. We selected this dataset for two reasons: (1) The dataset is currently supported by an active community and is continuously updated; (2) The dataset contains over 10,000 real ANDROID apps having different scope and characteristics and is more recent than others (it was released in November 2018).

[5]INSIDER repository: `https://github.com/insidersec/insider`.

[6]The KAGGLE dataset of ANDROID apps: `https://www.kaggle.com/lava18/google-play-store-apps`.

We first considered free and open-source apps from the initial set of applications. This was needed because of the requirements of the static analysis tools selected, which need to decompile the source code of the apps before detecting security threats. We considered the apps that could be freely analyzed to avoid incurring legal issues. Also, we only considered apps available on the *Google Play Store*. These two filters led to a dataset composed of 6,500 applications whose size ranges from 1MB to 99 MB, while the number of installs from a few dozens to 500,000 million. In addition, the apps were fairly equally distributed among all the categories of the GOOGLE PLAY store, meaning that we could analyze apps designed to deal with different objectives and targets.

RQ_1. A Taxonomy of Vulnerabilities Detected by Existing Static Analysis Tools for Mobile apps

While the description of the tools already indicates the security issues they can detect, the effort in this phase was needed to homogenize names and types of issues identified. Indeed, different tools could detect similar vulnerabilities but name them differently. The goal of **RQ_1** was to define a unique schema able to represent the issues identifiable with current static analysis tools.

Hence, we conducted iterative content analysis sessions [200] involving two software engineering researchers (1 PhD student, 1 faculty member) who have more than ten years of programming experience (the *inspectors*).

TAXONOMY BUILDING PHASE. Starting from the list of vulnerabilities detectable by the considered tools, each inspector independently analyzed each item and assigned it to a category based on both the OWASP official documentation and consulting online resources (e.g., papers, websites). Afterward, the inspectors opened a discussion and solved disagreements—this happened in 15 cases (out of 60 vulnerability types). The process led to the definition of a hierarchical taxonomy composed of two layers. The top layer consisted of 11 categories, while the inner layer contained 41 subcategories described in Section 9.3.

TAXONOMY VALIDATION PHASE. To reduce possible threats to conclusion validity, we decided to validate the defined taxonomy by involving two ANDROID security experts with 5 and 4 years of experience, respectively. These were contacted via e-mail. We provided the developers with a spreadsheet containing a list of 25 randomly chosen vulnerabilities from 41 security issues detected by the selected tools. The developers' task was to categorize the security issues according to the taxonomy previously built (which we provided in a PDF format). The developers could consult the taxonomy or assign new labels if needed. Once the external developers conclude the task, they send back the spreadsheet annotated with their categorization. As a result, both developers found the taxonomy clear and complete: they always assigned labels contained in the taxonomy without adding other categories.

RQ_2. On the Detection Capabilities of Existing Static Analysis Tools

The objective of **RQ_2** was to investigate the behaviour of the existing tools more closely. We executed them against the `apk` files of the considered apps and collected their output—to homogenize the output, we developed a parsing tool that converted the output of the tools into `csv` files.

The collected `csv` files were then used to address **RQ_2**. In this respect, we noticed that the tools failed to produce results in some cases. To maximize the number of applications analyzed, the first author attempted to fix the encountered issues manually (e.g., fixing links to external dependencies or changing some versions of some libraries). Nonetheless, we could not address the issues in 20%, 25 %, and 20% cases for ANDROBUGS2, TRUESEEING, and INSIDER, respectively.

For the remaining apps, we computed the number of vulnerabilities—classified according to the taxonomy coming from **RQ_1**—detected by the considered tools. This investigation provides a quantitative measure of the number of vulnerabilities detected by the tools. It provides insights into their detection capabilities concerning the various vulnerability types, hence potentially indicating the strengths and weaknesses of the tools.

Finally, we exploited the taxonomy built to evaluate the complementarity of the considered tools. We measured (1) the number of vulnerability types detected by more tools and (2) the number of vulnerability types solely detected by only one of them. In so doing, we considered both the levels of the taxonomy so that we could provide finer-grained insights.

9.3 ANALYSIS OF THE RESULTS

This section reports the results of our study. For the sake of clarity, we discuss each research question independently. Afterward, we discuss the overall findings of the study.

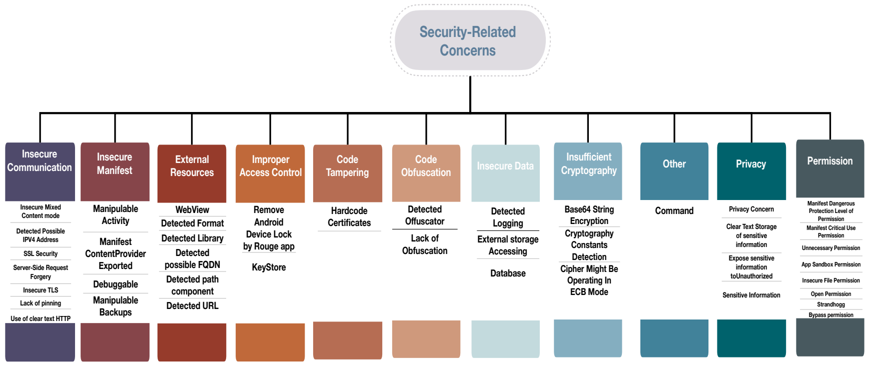

Figure 9.2: A Taxonomy of the Security-Related Issues Detected by the Considered Static Analysis Tools

RQ*₁. *A Taxonomy of Security-Related Issues Detected by Static Analysis Tools

Figure 9.2 overviews the taxonomy of security-related concerns identified by the considered automated static analysis tools. For the sake of space limitation, in the following, we only present the high-level categories of the taxonomy, while a complete description of the taxonomy, along with examples, is in our online appendix [259].

INSECURE COMMUNICATION - IC. This warning is generated when a client/server application exchanges information by means of inappropriate protocols [218], for instance, by relying on the *http* protocol. Developers could accidentally select insecure protocols to communicate with other applications or the environment; this might represent a security issue if sensitive data are exchanged.

INSECURE MANIFEST - IM. This category refers to possible issues connected to the insecure use of the `AndroidManifest`. In the manifest, developers declare the application's behavior, including the permissions required. The concern arises when developers accidentally miss the definition of app restrictions, allowing the app to be potentially called by external malicious apps.

EXTERNAL RESOURCES - ER. This issue might arise in cases where mobile applications rely on external resources without putting in place any control over them. For example, the unchecked user or environment inputs represent a threat to security, as malicious users might format the input to create concerns for security.

IMPROPER ACCESS CONTROL - IAC. The application does not apply (or only partially applies) mechanisms to restrict access to resources from an unauthorized user. When those mechanisms are not correctly applied, other users can read sensitive information and execute commands [51]. An example of a successful attack is represented by the case of *Keystore*, which is a private repository that developers use to store sensitive or reserved data. A vulnerability affecting the API allowed malicious users to bypass permissions, leading to privilege escalation without user interaction [359].

CODE TAMPERING - CT. Code tampering is the process conducted by a malicious user to change the app's behavior or the APIs it relies on [60]. An automated static analysis tool could detect this issue when developers implement a potentially untrusted third-party library or other external application whose credibility cannot be verified. An example of this issue concerns the presence of hard-coded certificates, as a mobile app might run without verifying the external component it relies on.

CODE OBFUSCATION - CO. The category refers to the lack of code obfuscation. Developers apply this operation to make it harder for a hacker to access the source code. If code obfuscation is not applied, an attacker could read the source code by decompiling the apk, obtaining the respective Java code in case of native applications, and identifying vulnerabilities to exploit.

INSECURE DATA - ID. This threat predominantly refers to the data storage and occurs when developers assume that malicious users or malware applications will not have access to the file system. For this reason, they adopt insecure mechanisms to archive private data. Nonetheless, static analysis tools may detect an issue since an user still can perform a root procedure, through which s/he can have access to the entire system and break these mechanisms, hence allowing malware or external attackers to exploit the vulnerability and have access to sensible data [217].

INSUFFICIENT CRYPTOGRAPHY - ICR. This threat relates to the insufficient mechanisms adopted to preserve personal information [219]. In these cases, developers apply protocols to preserve personal information or sensitive data; however, these might be insufficient (e.g., if they select an insecure protocol to encrypt data).

PRIVACY - PR. This category refers to other generic privacy issues detected by the automated static analysis tools. In these cases, the tools do not explicitly label the vulnerability issues nor provide information to characterize them. However, from our iterative content analysis, we could realize that these unlabeled issues relate to some form of privacy concerns. For instance, ANDROBUGS2 detects issues when developers use inappropriately the so-called *Android_ID*, i.e., the ANDROID user IDs.[7] This might represent a security issue since, if externals get access to the user data, they can steal them [24].

[7]The *Android_ID*: `https://developer.android.com/training/articles/user-data-ids`.

PERMISSION - PE. The ANDROID ecosystem uses permissions to guarantee secure access to resources or protect users' privacy. Static analysis tools may detect permission-related issues when developers use weak permissions i. e., permission settings that are improperly configured too lenient or more than needed, potentially compromising users' privacy. For instance, an issue is detected if an app requests permission to access the storage but does not use it. In this case, a malicious user could exploit the vulnerability to read data in the user's storage.

Key findings of RQ$_1$.

The considered static analysis tools can detect 11 different types of security-related concerns, including improper platform usage and code obfuscation. In addition, we could identify 7 warning types that do not have a clear reference but relate to the categories Permission and Insecure Communication.

RQ$_2$. On the Detection Capabilities of Existing Static Analysis Tools

Our second research question was concerned with understanding the detection capabilities of the tools in terms of frequency of security issue detection, and complementarity among the considered tools.

FREQUENCY OF DETECTION. When determining the frequency of security-related concerns, we analyzed the output of the tools, mapping the vulnerabilities identified onto the taxonomy built in the context of **RQ$_1$**. The results were as follows.

ANDROBUGS2. Figure 9.3 shows the top-10 vulnerabilities detected by ANDROBUGS2. In almost 50% of the cases, the tool identified *'Web View'* and *'SSL Security'* vulnerabilities: these pertain to the *'External Resources'* and *'Insecure Communication'* categories of the taxonomy. Looking at the figure, we could then observe that ANDROBUGS2 could identify various types of vulnerabilities related to different security concerns. While these were detected in fewer cases, the tool seems to support

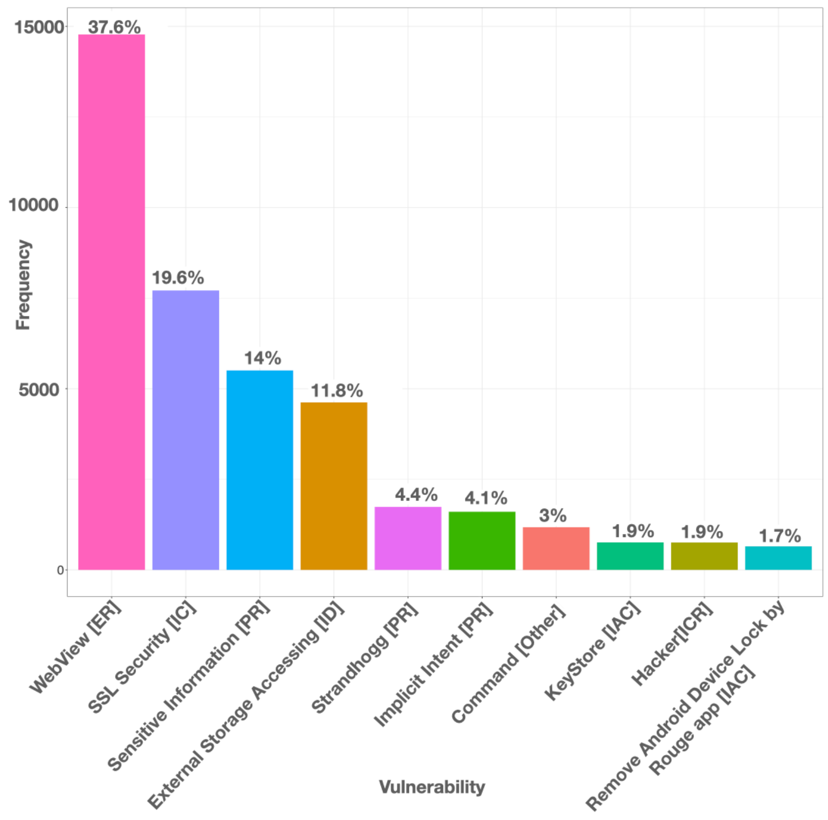

Figure 9.3: Top 10 vulnerabilities detected by AndroBugs2.

developers in detecting many vulnerabilities. In addition, AndroBugs2 could identify at least one instance of each vulnerability of the taxonomy. On the one hand, further experiments aiming at assessing the accuracy of the insights provided by the tool would be needed. On the other hand, the fact that the tool could detect such a wide range of vulnerabilities might provide indications on the health status of mobile apps, which may be particularly exposed to security issues that threaten their users.

TRUESEEING. Figure 9.4 reports the ten most frequent vulnerabilities detected by TrueSeeing. As shown, almost 39% of the vulnerabilities found by the tool are connected to the use of logging files, which fall under the *'Insecure Data'* category. While logging is typically considered

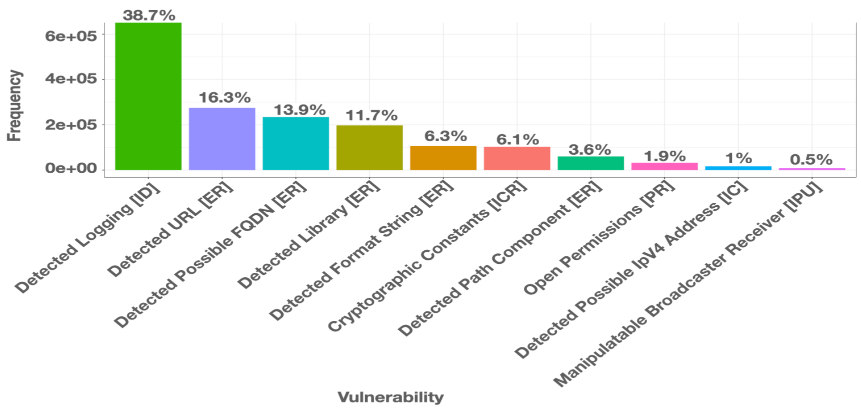

Figure 9.4: Top 10 vulnerabilities detected by TRUESEEING.

a best practice [391], in some cases, developers log sensitive information, e.g., sensitive keys or URLs. As a consequence, an attacker could potentially exploit logs to damage the app. In fewer cases, TRUESEEING identified security-related concerns related to the *'External Resource'* category, such as *'Detected URL'* (16%) and *'Detected Possible FQDN'* (14%). Both vulnerabilities make data available to externals, namely URLs in the former case and Fully Qualified Domain Name (FQDN) in the latter. Other vulnerabilities were detected to a lower extent. Looking at the categories of those vulnerabilities, we can say that TRUESEEING identifies a variety of problems, ranging from cryptography to permission issues. Code tampering problems represent the only exception: this is the vulnerability that the tool was unable to detect in our dataset. At the same time, the set of security-related concerns the tool identifies is quite different from those observed with ANDROBUGS2, suggesting a possible complementarity between the two.

INSIDER. The behavior of INSIDER was drastically different from the one of the other tools. In this case, we could detect only two categories of vulnerabilities, namely *'Exposed to sensitive information to an unauthorized actor'* (61% of the warnings pertained to this vulnerability) and *'Clear text storage of sensitive information'* (39%). Both of them fall under the *Privacy* category of our taxonomy and have to do with sensitive information inappropriately stored within the context of a mobile app. The observed behavior of the tool suggests that it has a close focus on privacy issues, while other vulnerability categories cannot be frequently identified. This is likely the characteristic that makes the tool different from the others considered. In addition, it is important to remark that, differently from the claims made in the official documentation, we could not find any reference to the detection of vulnerabilities listed by OWASP. This likely suggests that the documentation available is outdated.

Summing up, different tools seem to focus on different categories of vulnerabilities. It is also worth remarking that we could identify a number of security-related categories that are only rarely detected by the tools. Comparing the categories composing our taxonomy (see Figure 9.2) with those detected by the tools, we can indeed report that major security categories, like *Code Tampering*, *Improper Platform Usage*, and others, which have been the subject of previous studies in literature [106, 205], are poorly identified by the considered tools. Of course, this might be due to the fact that certain types of vulnerabilities are poorly diffused in ANDROID apps. However, we still point out the need for additional experimentations to assess the support provided by current tools.

Key findings of RQ$_2$ - Frequency.

Different tools detect different security-related concerns with different frequencies. Certain categories of problems are (almost) never detected, possibly suggesting the need for further studies on the actual support provided by the considered tools in practice.

COMPLEMENTARITY AMONG THE TOOLS. From the frequency analysis, we discovered that different tools seem to capture different security-related concerns. With our last analysis, we sought to provide additional insights into the complementarity among the tools. We could first observe that INSIDER does not capture any category of problems that the other tools cannot already detect. In this sense, the tool seems to provide fewer benefits in practice, as it not only identifies a lower amount of security concerns but also targets problems that other tools can detect—of course, these observations should be backed up with additional analyses on the accuracy of the recommendations provided to developers. As for AndroBugs2 and Trueseeing, instead, we could observe that the two tools cover pretty different categories of problems. Indeed, only the category of *'Insecure Data'* is in common. These findings suggest that the tools might be combined by developers to enlarge the coverage of security-related concerns.

Similar conclusions could be drawn when considering the security concerns about the second level of the taxonomy. Analyzing those issues, we could discover that only *'Web View'* and *'Manipulable Activity'* vulnerabilities are in common between AndroBugs2 and Trueseeing.

Key findings of RQ$_2$- Complementarity.

The combination of AndroBugs2 and Trueseeing may provide more extensive coverage of security-related problems. On the contrary, Insider does not cover vulnerabilities that are not detected by the other tools.

9.4 THREATS TO VALIDITY

THREATS TO CONSTRUCT VALIDITY. One of the objectives of this chapter was concerned with the categorization of the security-related concerns detected by existing automated static analysis tools. This objective was naturally threatened by the selection and accuracy of the considered tools. Our choice was driven by multiple considerations. In the first place, we have considered only open-source static analyzers that can be called via command line and that take an `apk` file as input. The set of tools analyzed restricts our possibility of providing a comprehensive view of the issues detected by currently available tools. From the accuracy of the tools selected, our work might be threatened by not considering *false positives* and/or *false negatives*. The identification of *false positives* was not possible in our case. This indeed requires knowledge of the application domain and the source code of the considered apps: as such, the identification should have been done by the original developers of the application, but, unfortunately, this was not feasible. For this reason, our results might be partially affected by the presence of *false positives*. At the same time, our analysis could not deal with *false negatives*, namely the actual security-related concerns that the tools did not identify. Of course, this limitation of our study cannot be addressed as the tools could not detect those issues. Our analysis sets a *lower bound* for researchers and practitioners: despite the inherent limitations, our findings still quantify how the currently available vulnerability and malware detectors support developers. Researchers interested in further elaborating on the matter may build on top of our findings, providing an improved view of the capabilities of the tools.

THREATS TO CONCLUSION VALIDITY. The major threat in this category concerns the research methodology used to address the research questions. When considering **RQ**$_1$, we applied an iterative approach to extract the taxonomy of security-related concerns detected by the existing tools. This task requires a human-intensive effort, is subjective by design, and leads to wrong interpretations. To mitigate this threat, we used (as possible) a systematic approach and two inspectors participated in building the taxonomy. In **RQ**$_2$, a possible threat refers to the different granularity of the vulnerabilities identified by the considered tools. Part of our future work will investigate the impact of this aspect on our findings more closely.

THREATS TO EXTERNAL VALIDITY. There are four important considerations. First, we considered three tools, meaning that the coverage of the conclusions is somehow limited. Secondly, we decided not to consider dynamics or hybrid approaches to detect possible vulnerabilities. This was a methodological decision: we were focused on analyzing statics vulnerability tools, so other kinds of methods were considered out of scope. Third, our study considered 6,500 ANDROID apps, and it can be regarded as one of the most extensive empirical investigations up to date

[37, 364]. Finally, the empirical investigation considers ANDROID apps: as such, our work might not generalize to apps written using different platforms. Future replications of this work are desirable: to enable them, we released a replication package that would allow further researchers to reproduce our study in other contexts.

PART IV

DISCUSSION, FURTHER ANALYSIS, AND CONCLUSION

10

DISCUSSION AND IMPLICATIONS

This chapter examines the main findings identified in previous chapters. To improve the readability, we will discuss the results achieved to respond to our RG, and then, we will discuss the results obtained by further studies.

10.1 RG. CODE QUALITY IN SOFTWARE SYSTEMS OVER TIME

We addressed the RG_a in Chapters from 3 to 6. The results provide a number of reflections, implications, and actionable items for researchers and practitioners that need to be elicited.

RG. Reflections, Implications, and Actionable Items

REUSABILITY MECCHANISMS ON CODE SMELLS. Our findings partially contradict most of the investigations previously done on the matter. As already mentioned, a number of empirical investigations established a negative relation between the adoption of reusability metrics and code complexity [8, 9, 122]. However, the evolutionary nature of our study revealed something different: in most cases, the correct adoption of inheritance and delegation positively related to the decrease of code smell severity. While we are aware that further analyses would be needed to confirm our conclusions on a larger scale, the results obtained so far allow practitioners to (re-)consider reusability as a core mechanism for evolving software systems. Similarly, our findings suggest that broader and perhaps more conclusive indications into the usefulness of code metrics might be obtained by looking at how these metrics evolve over time and how they impact code quality.

Design patterns are intended to improve the quality of code, as they are introduced with the purpose of organizing the classes in such a way that good attributes of inheritance and delegation are enhanced. However, implementing such reuse mechanisms is not trivial, as developers may accidentally fall into mistakes, such as overuse, e. g., adding design patterns when not necessarily needed, or misuse, e. g., making a sub-optimal choice on the kind of pattern to employ for a specific problem. Such mistakes can eventually lead to bad effects, unnecessarily adding atoms of complexity to the code, which in turn can result being less comprehensible and even affected by smells, as observed in our study. Therefore, there is the need for developers to focus on properly applying design patterns [394], to avoid declining their positive effects into worsening the code quality.

INHERITANCE AND DELEGATION ON DEFECT-PRONENESS. In this respect, two observations should be made. In the first place, we discovered that inheritance and delegation metrics, coming from the operationalization of the reusability mechanisms used by developers, have a relatively low impact on defect-proneness. In the second place, we found out that the control variables of our statistical analysis, namely the metrics pertaining to the Chidamber & Kemerer [68] metric suite, have also a limited connection to defect-proneness. Both findings are somehow surprising: these metrics were indeed experimented in plenty of studies on source code quality and researchers have been often analyzing the extent to which they can support the monitoring and prediction of defect-proneness of source code [38, 128].

To provide further, more actionable insights into our findings and better understand the extent to which our statistical analysis would be actually corroborated when considering the impact of code quality metrics on defect prediction.

More specifically, given that our analysis granularity level was the commit and that we needed to account for the time relations between commits, we focused on the so-called *just-in-time* defect prediction [163], that is, the creation of defect prediction models able to assess the defectiveness of individual code commits based on the data collected through the analysis of previous commits.

To make our analysis as precise and sound as possible, we conducted a partial replication of the work by Pascarella et al. [271], who experimented with a large set of features composed of 24 process, product, and developer-oriented metrics to capture the defectiveness of code commits. As product metrics, the original authors used the metrics also employed within our study. Through this replication, we could therefore assess the role of these metrics when considering their contribution to defect prediction, other than comparing such a contribution with respect to additional metrics typically used in defect prediction, hence enlarging our overview on the value of the considered metrics. While Pascarella et al. [271] mainly focused on a variant of the problem of just-in-time defect prediction aiming at predicting defective files within commits rather than defective commits, they also compared against a standard just-in-time defect prediction model, hence enabling an analysis at commit-level. The reason for relying on this work was twofold. In the first place, Pascarella et al. [271] released an online appendix with all the scripts used in their study and documentation that enables the exact replication of their work: as such, we avoided possible bias due to the re-implementation of the defect prediction model. Second, Pascarella et al. [271] took into account a large amount of metrics having different nature and coming from previous literature on defect prediction [163, 288]: as such, we could conduct a larger and sound experimentation of how quality metrics affect the performance of just-in-time defect prediction. To conduct our analysis, we performed the following steps:

- For each project considered in our study, we mined all the commits to compute the 24 process, product, and developer-oriented metrics. Since the metrics were computed on the files modified within the considered commits, we aggregated them to have a unique commit-level value for each metric. This was done using the "group by" operation, considering the commit hash as the primary key, and applying the mean and median over all the metrics;

- We merged the information collected with the one available in our dataset: for each project and for each commit, we combined the 24 process, product, and developer-oriented metrics with the inheritance and delegation metrics;
- We trained and tested a *Random Forest* classifier, i.e., the best classifier identified in the work by Pascarella et al. [271], by applying a Time Series Split validation. This is a time-aware variant of the cross-fold validation that (i) divides the dataset into K (in our case, K = 10) folds and (ii) in the kth split, it returns first k folds as train set and the (k+1)th fold as test set.[1]
- This validation can be applied when the time order may impact the results and avoid training the model using future commits to predict the defectiveness of past commits. The performance of the model was assessed through multiple evaluation metrics such as precision, recall, F-Measure, and AUC-ROC.

We investigated two predictive configurations. In the first one, we devised a *binary* defect prediction model that predicts a commit as defective or not, i.e., the standard defect prediction scenario. In the second configuration, we devised a *multi-class* defect prediction model able to assess how the source code defectiveness varies over the evolution of the project, i.e., a defect prediction scenario where the task is to foresee the defectiveness trend in terms of increase, decrease, or stability of the number of defects within a software project. This latter scenario is closer to the research methods employed in our study and was set up with the aim of embedding additional evolutionary considerations within the defect prediction model and investigating the contribution of code quality metrics to assess the overall defectiveness of a software project. From a more technical perspective, the model was devised to assign a commit

[1] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TimeSeriesSplit.html

to a categorical variable within the set {*'Increased'*, *'Decreased'*, *'Stable'*}, namely the same variables used within the Multinomial Log-Linear statistical model.

For both scenarios, we executed the model in two runs: the first utilizing all metrics and the second excluding metrics related to inheritance and delegation. This approach allowed us to closely observe the effects of the key variables in our study, namely inheritance and delegation metrics. By doing so, we aimed to quantify the accuracy improvement or deviation when these metrics were included as features in the defect prediction models. Additionally, we calculated the importance of each feature to determine which metrics were most influential for the models tested.

In terms of results, we could draw multiple considerations. When considering the binary defect prediction scenario, the performance achieved was close to 94% in terms of F-Measure both when considering the models with and without inheritance and delegation metrics. On the one hand, this result seems to indicate that the overall defect prediction capabilities cannot be improved through the use of reusability metrics, hence confirming our results, i. e., inheritance and delegation metrics have a limited connection to defect proneness. On the other hand, it is worth observing that improving over an F-Measure of 94% is always particularly tough: in this sense, the contribution given by inheritance and delegation metrics may be somehow "hidden" by the high performance of the defect prediction model. As a consequence, a more reasonable way to assess the contribution of reusability metrics was to assess the feature importance of the metrics considered by the model relying on inheritance and delegation indicators. Through this analysis, we discovered that (1) the *Random Forest* classifier never selects *specification* and *implementation* inheritance among the top-20 features to use for predicting defective commits in the considered projects; (2) the amount of delegations was in the top-15 features employed by the model in all the projects; (3) the specification and implementation inheritance metrics had limited predictive power, with other inheritance metrics such as NOC and DIT having a slightly higher impact on the predictions. These findings were perfectly in line with our observations: we could indeed further corroborate that the defect-proneness of source code is only partially dependent on reusability metrics and that, instead, the way developers structure hierarchies might impact defects more than the specific reusability mechanisms employed.

Our findings also revealed that the control variables used in our statistical analysis, i. e., the Chidamber-Kemerer metrics, were not statistically impactful on defect proneness. The defect prediction investigation confirmed these findings as well. Indeed, the feature importance analysis constantly reported process metrics such as the entropy of changes [137], the scattering of code changes [84], and commit date [288] as the most impactful features. In the first place, our findings corroborate previous research showing that process metrics can better predict defects with respect to traditional code quality attributes [288] and, as a consequence, provide additional support to the research field involved in the definition of process and developer-oriented metrics for defect prediction. Secondly, our research outlines that the use of code quality metrics, including the inheritance and delegation ones, to assess the defectiveness of source code may result in suboptimal recommendations for developers and, for this reason, these metrics should be used for different purposes and/or for different use cases: e. g., our previous work [118] revealed that quality, inheritance, and delegation metrics can positively contribute to the evolutionary analysis of code smells.

A similar discussion could be done when considering the multi-class prediction model. Also in this case, we found that the models relying and not on reusability metrics had similar performance in terms of F-Measure (94%), with inheritance and delegation metrics that were selected by the *Random Forest* classifier for all projects. While they had a lower predictive power than NOC and DIT, we found that both inheritance and delegation metrics were more impactful than cohesion, coupling, and complexity metrics, i. e., LCOM, CBO, WMC. As such, we could further corroborate that quality, inheritance, and delegation metrics have a limited connection to defect proneness. Similarly to the previous experiment, the entropy of changes [137], the scattering of code changes [84], and commit date [288] were the most important characteristics to predict defective commits, hence suggesting that evolutionary considerations on the defect proneness of source code should be made through the analysis of historical information coming from the complexity of the development process.

All in all, our findings corroborated the negative results obtained by previous researchers who experimented with code quality metrics in defect prediction [141, 161, 287]. While this is already worrisome for the entire software maintenance and evolution research community, our findings should be considered as even more worrisome because of the granularity of the analysis conducted. We indeed elaborated on the change history information of software projects, analyzing how code quality metrics were related to defect-proneness throughout the evolution of the considered projects, discovering that none of them was statistically correlated to the variation of defect-proneness. As such, our results represent an additional alarm signal for the research community.

☛ *Implication 1. An improved understanding of the role of code metrics for source code quality can be obtained by looking at their evolution and how these impact code quality attributes. As such, the research community might consider novel empirical investigations aiming at characterizing the long-term, evolutionary impact of code metrics.*

☛ *Implication 2. Practitioners need to be informed on the benefits and drawbacks of reusability mechanisms during maintenance and evolutionary tasks.*

MONITORING USAGE TRENDS TO IMPROVE SOFTWARE QUALITY. Altogether, these findings seem to suggest that an advanced knowledge on how to improve software quality might be obtained by exploiting precious pieces of information coming from the analysis of the change history of software projects. For instance, we envision the definition of monitoring techniques that, by exploiting the way developers use

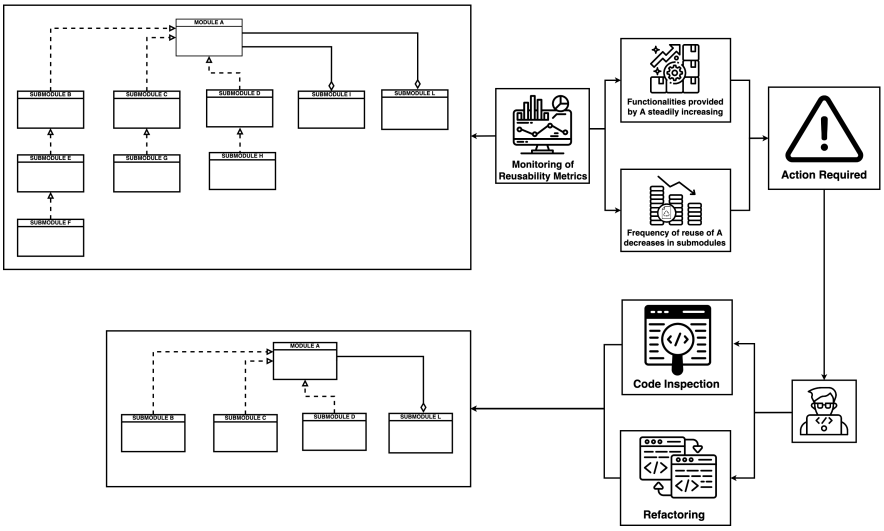

Figure 10.1: Use case scenario in which the monitoring of reusability metrics might be exploited.

to adopt code reusability mechanisms, may recommend the most appropriate actions to conduct while performing corrective maintenance. Similarly, we can envision the definition of novel approaches based on nudge-theory [48] to stimulate developers toward the more frequent or most appropriate adoption of code reuse to reduce the overall defect-proneness of source code. To make our conjectures more tangible, let us consider the scenario depicted in Figure 10.1, which represents the way we envision a monitoring system may support developers during software maintenance and evolution. More specifically, suppose that a system 'S' contains a module 'A' having (1) multiple submodules, i.e., 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I' and 'L' in Figure 10.1, each either directly or indirectly inheriting from 'A'; (2) some operations through which the submodules delegate operations to 'A'. In such a scenario, a regular monitoring of reusability metrics or the prediction of usage trends may allow the developer to observe or predict the way the inheritance and delegation relations vary over time, possibly detecting or even preventing the increasing complexity affecting 'A' and its submodules, other than the presence of suboptimal design decisions that would require some refactoring actions.

For instance, suppose that in the scenario proposed in Figure 10.1 a monitoring system realizes that the amount of functionalities provided by 'A' is steadily increasing, with the frequency of 'A' being reused decreasing in the submodules—this case may indicate that the system is in the descending path of a *'increasing-decreasing'* implementation inheritance pattern. This may indicate a suboptimal use of inheritance and delegation: 'A' offers more services, but the submodules inheriting from it do not fully exploit them, suggesting that they are not properly exploiting the inheritance mechanism—note that a similar scenario has been associated with multiple risks for software reliability, including an increasing change- and defect-proneness [265] and a higher likelihood of the system being maliciously attacked because of the suboptimal visibility granted to fields and operations [328]. By monitoring reusability metrics, multiple insights may be provided. On the one hand, developers may be informed of the evolution of reusability metrics and exploit such an information to schedule quality assurance sessions aiming at reducing quality and security concerns, e.g., code review targeting 'A' and the way the submodules interact with it. On the other hand, automated instruments might exploit reusability metrics to recommend refactoring actions aiming at simplifying the hierarchy: for instance, the situation described above, i.e., submodules not fully exploiting the features of 'A', may suggest the presence of a *Refused Bequest* smell [105], whose refactoring may either consist of defining a new superclass only containing the fields and operations that are actually needed to the submodules, i.e., *Extract Superclass* refactoring, or replacing the inheritance mechanism with delegations, *Replace Inheritance with Delegation* refactoring.

Based on the considerations above, the multifaceted ways our findings can be exploited represent a call for researchers in software quality.

REUSE AND ITS ADOPTION: TWO SIDES OF THE SAME COIN. Our empirical investigation revealed a dichotomy between the concept of code reusability and its actual application. In particular, we found that while reusability itself is a useful instrument to improve software quality and reduce maintenance effort, an inappropriate adoption of these mechanisms may have negative implications. This is indeed the case observed with DIT and NOC in our statistical exercise, two well-known metrics that measure the extent of the hierarchical relations among classes. We found that increases in terms of hierarchical relations lead to negative variations of the defect-proneness of software artifacts. As such, we argue the need for further research, especially in terms of software refactoring optimization. Researchers are indeed called to better investigate the reasons behind the misuse of inheritance and delegation mechanisms and when and why these can deteriorate software quality. These investigations would be instrumental to the definition of novel refactoring techniques that may support developers while optimizing hierarchies of classes.

At the same time, our findings provide two key implications for practitioners. On the one hand, an improved knowledge of the usage patterns might be beneficial to understand the way code reusability evolves in their own projects: practitioners would therefore put in place monitoring instruments to verify the evolution of inheritance and delegation uses and assess how the usage trends co-evolves with software quality. On the other hand, our results might be exploited by practitioners to reason on the use and misuse of inheritance and delegation mechanisms, other than on how the creation of complex hierarchies might possibly worsen source code quality and increase corrective maintenance effort.

PREDICTION OF CODE QUALITY PROPERTIES: THE ROAD AHEAD. Another aspect to consider is the one concerned with the prediction of code quality properties. In this respect, the findings coming from our research questions altogether contribute to increase the research community awareness with respect to the need for novel code quality prediction techniques and tools. First, the traditional code quality metrics employed in prediction models have little to no correlation to defect-proneness. Second, code reusability mechanisms might potentially boost the code quality analysis and possibly being used within predictive modeling techniques. In addition, the usage trends can be exploited to recommend which of the features would be more worth to use in specific moments of the evolution. All these aspects, emerged from our analyses, represent future perspectives that our research community would like to further investigate. We envision multiple experimentations aiming at revisiting previous findings obtained in literature to account for the evolutionary nature of software - the research method employed in our study, which took the change history information into account, may indeed be generalized to understand how different code quality metrics evolve over time and how they impact software quality. In our opinion, analyses of this type would potentially lead to revolutionize code quality as we know, revealing insights driven by the actual adoption of code metrics by developers.

Contemporaneous, we envision novel techniques that, by analyzing the evolutionary development context, may feed predictive models with the most relevant metrics to predict source code quality. Also in this case, we believe that an evolution- and context-aware view of predictive software maintenance might substantially boost the support that we, as researchers, may provide to practitioners.

☛ *Implication 3. The SE community needs to conduct further research to identify more representative metrics for measuring code quality.*

CODE SMELLS IN AI-ENABLED SYSTEMS. Based on our results, developers need to be informed of the potential impact of code smells on their systems during the software evolution process. Despite the existing body of research that emphasizes the importance of monitoring quality attributes to prevent a subsequent increase in effort [23, 323], there is still a need for further empirical investigations to determine the extent to which code smells can pose a danger.

☛ *Implication 4. More empirical research needs to be done to begin to make developers informed of the potential issues associated with the presence of code smells.*

DIFFERENT MINDSETS IMPLY DIFFERENT SMELLS. Python developers often employ specific coding constructs to avoid loops or reduce the amount of code needed for tasks, which can lead to unique code smells not typically found in other languages like Java. These smells, although sometimes differing from those documented in the literature, can still degrade software quality over time. Therefore, a detailed examination of Python's distinctive features is necessary to determine best practices that the Python community should follow to minimize these code smells in their systems.

☛ *Implication 5. A thorough analysis of the Python community is needed to understand whether what they consider best practices may be antipatterns that can cause code smells.*

Key findings of RG.

Our findings indicate that reusability mechanisms contribute to reduced code smells and positively affect other important software maintenance aspects, such as the effort required for defect resolution. The correct adoption of inheritance and delegation is associated with a decrease in code smell severity. In contrast, reusability mechanisms employed by developers exhibit a relatively low impact on defect-proneness. The control variables in the statistical analysis, particularly metrics from the Chidamber & Kemerer suite, show limited connections to defect-proneness. We discovered that an increase in hierarchical relations is linked to negative variations in the defect-proneness of software artifacts. Furthermore, design patterns are not, in any case, the “best solution” to increase code quality, indeed, implementing these mechanisms requires careful consideration, as developers may inadvertently make mistakes such as overusing or misusing design patterns. These mistakes can result in added complexity, making the code less comprehensible and prone to the emergence of smells.
Lastly, code smells in Python Systems represent a critical issue for developers. Indeed, the built-in functions that Python provides induce the proliferation of specific code smells; our results also suggest that most times, Python developers introduce code smells in their systems due to evolutionary activities.

10.2 ADDITIONAL CONSIDERATIONS: FURTHER STUDIES

We addressed the further studies from Chapters 8 to 9. The results provide implications for the Software Engineering community, which we will explain in the next sections.

Further Studies: Reflections, Implications, and Actionable Items

Our studies provide a number of observations of how code quality declined in emerging systems. Based on previous chapters, we discovered that the software engineering community shyly investigated emerging systems by considering code quality aspects.

The following sections describe the main reflections and implications identified for these systems.

ON VERIFIABILITY AND REPLICABILITY. As noticed throughout our systematic literature review and analyses, a large number of primary studies do not report granular information to enable neither verification nor replication. In addition, the papers are rarely accompanied by replication packages that make data and scripts publicly available for other researchers. In our humble opinion, this represents a key threat to dissemination and verification of the published research papers, which leads to two implications concerned with the way research papers are disseminated:

☛ *Implication 1. Researchers should consider including more methodological details to enable an improved understanding of the design and definition of the proposed techniques. This would be beneficial in terms of dissemination, as practitioners might better understand how to put the defined techniques into practice, increasing the overall impact of the research on the matter. At the same time, this would support research, since additional investigations might be made on top of the findings achieved by previous researchers, further increasing the impact of research.*

AI & IOT PRIVACY: THE ROAD AHEAD FOR SE4AI RESEARCH. The results to our research questions clearly indicate that there is still a long and winding road to making artificial intelligence suitable for the problem of IoT privacy. According to our analyses, this pertains to several aspects that, in turn, call for several implications for software engineering for the artificial intelligence research community:

☛ *Implication 2. Researchers have been mainly focusing on six tasks: i) User Authentication, ii) Network Analysis, (iii) Attack Detection, (iv) Framework Building, (v) Malware Detection, and (vi) Privacy-Preserving Scheme. At the same time, our systematic search identified several other tasks that have received less attention and that further research might consider. Perhaps more importantly, we highlight the lack of insights from the trenches, namely the lack of empirical investigations that target the practitioners' and IoT users' perspectives and might reveal other relevant tasks that the current body of knowledge has neglected. Consequently, we claim that the first relevant aspect for future research in the field is represented by a large-scale analysis of IoT privacy in practice.*

☛ *Implication 3. A third, critical issue unveiled by our work relates to the public datasets currently available. Besides having only a few datasets to experiment with, the major criticism is concerned with the level of realism and actual suitability of these datasets. As commented in Chapter 8, some datasets are unbalanced, being potentially unsuitable for training artificial intelligence techniques. Moreover, the most widely used datasets are biased toward certain accuracy indicators, hence biasing the interpretation of the results. This is, likely, the most important issue encountered by our systematic review, as it impacts most research conducted so far. Therefore, we argue that concrete steps should be conducted to devise novel, more reliable datasets to re-assess the experiments performed so far. We hope that the indications provided by our work, in terms of limitations and challenges of the methodology employed by current papers, might help design better the empirical investigations into the performance of artificial intelligence techniques.*

ON THE SUPPORT PROVIDED BY STATIC ANALYSIS TOOLS. We conducted an analysis aiming at comparing the support of the tools against the list of the top vulnerabilities identified by the Open Web Application Security Project (OWASP), one of the main security foundations worldwide that periodically produces reports about the most frequent and harmful mobile vulnerabilities.[2] From this analysis, we could observe that the current tools only partially align with the OWASP mobile top-10. While the detection of security-related concerns classified as *'Improper Platform Usage'*, *'Insecure Communication'*, or *'Insufficient Cryptography'* is supported by some of the considered tools, e.g., ANDROBUGS2, a number of other critical issues are still neglected. For example, the categories of *'Client Code Quality'* or *'Extraneous Functionality'* are not considered by any tool. In addition, it is also worth mentioning that the OWASP top-10 considers the issues pertaining to *'Improper Platform Usage'* as the most popular nowadays. Nonetheless, our frequency analysis revealed how this category is not among the most frequently identified, possibly indicating the inability of the tools to deal with this category of security concerns. In other terms, there seems to exist a mismatch between what the tools detect and what they should provide support for. We argue that this mismatch should be further investigated by our research community and tool vendors, which might be interested in providing additional support for ANDROID developers.

☛ *Implication 4. The software engineering community needs to focus on the improvement of existing vulnerability detection tools to assist developers during vulnerability detection tasks.*

ON THE ACCURACY OF CURRENT TOOLS. In our study, we executed three state-of-the-art tools on a dataset of ANDROID apps, analyzing their output from various perspectives. We recognize that our observations might be threatened by the presence of false positives, i. e., wrong indications given by the tools. Yet, our empirical setting allows us to highlight the lack of empirical investigations into the accuracy of static analysis tools for ANDROID apps. Perhaps more worrisome, we point out the lack of datasets that might be used for this purpose. Therefore, we call for more research on the matter and the definition of novel datasets that can be exploited to compare and improve the current state of static analysis in mobile applications.

[2]The OWASP Mobile Top-10: https://owasp.org/www-project-mobile-top-10/
.

☛ *Implication 5. Researchers must release updated datasets on vulnerability to improve the existing static analysis tools.*

ON THE COMBINATION OF MORE TOOLS. The results coming from Chapter 9 allowed us to assess the complementarity among the three tools, which revealed that different tools might identify different security concerns. We may argue that combining multiple tools can achieve more extensive coverage of the issues affecting a mobile app. On the one hand, this finding can be exploited by practitioners, who might be willing to adopt and run more tools against their code. On the other hand, such a complementarity might be an opportunity for researchers who might want to devise novel smart techniques able to automatically combine static analysis tools based on the context or the developer's needs.

ON THE SECURITY OF MOBILE APPS. While our analysis's main goal was establishing the current support provided by static analysis tools, it also revealed insights into the security of the mobile apps analyzed. Our frequency analysis indicated that mobile apps are frequently affected by security-related concerns. Therefore, it would be useful to better analyze the current state of security in Android. For instance, empirical investigations targeting the vulnerabilities detected by the tools to elaborate on the reasons behind their introduction might provide key insights for practitioners interested in improving the security profile of their apps.

☛ *Implication 6. Practitioners who daily work on mobile apps need to be informed on the possible inefficiency of static analysis tools to detect vulnerability in their applications.*

Key findings of further studies.

The results achieved for our further studies indicated that AI techniques are usually unsuitable for dealing with privacy in a real-world scenario. In addition, we discovered that most of the time, the artificial intelligence techniques adopted by researchers work in agnostic contexts, suggesting that their applicability does not depend on the environment where IoT devices work.
Finally, our investigation reveals that vulnerability tools only partially cover the top-10 OWASP, showing a mismatch between the quality issues detected and the support needed. This suggests the potential benefit of combining multiple tools for comprehensive coverage in addressing security issues in mobile applications.

11 FURTHER ANALYSIS

11.1 FURHTER ANALYSIS AND NEXT DIRECTIONS

This chapter explores further analyses and the research directions that need to be explored to comprehend how code quality needs to be investigated in software systems.

LIMITED TOOLS AND RESEARCH INSTRUMENTS. While the literature and studies in this dissemination analyze code quality from various perspectives, there is a need for tools to examine and address this issue comprehensively. As explained in Chapter 3 and Chapter 4, we developed a static analysis tool tailored for detecting programming abstractions in Java source code. Nevertheless, this tool represents an initial step to addressing code quality concerns. The broader spectrum of software reusability mechanisms warrants exploration beyond programming abstractions and design patterns, encompassing other facets such as third-party libraries.

It's worth noting that current tools are limited to analyzing Java source code, which restricts the ability to investigate AI-enabled systems, typically developed using Python. These systems heavily rely on third-party libraries, potentially leading to quality issues stemming from misuse or abuse of such mechanisms. Therefore, there's a pressing need to extend analytical capabilities to encompass Python and scrutinize the quality implications of utilizing third-party libraries within these contexts.

Research Directions:

- 👉 The software engineering community needs to release tools to detect reusability mechanisms for other programming languages.
- 👉 Investigate the quality implications of utilizing third-party libraries in Python codebases. This could involve studying common issues encountered when using these libraries and developing strategies to mitigate potential risks.
- 👉 Investigate strategies and develop tools to support code quality analysis in multi-language codebases, particularly those involving both Java and Python. This could include techniques for interoperability between existing analysis tools or the development of new tools capable of analyzing code written in multiple languages.

PARADIGM SHIFT. Based on the results obtained from Chapters 8 to 9, the adoption of systems developed using emerging technologies drastically increases over time. These results suggest that a *paradigm shift* is required for these systems, i. e., traditional software paradigms (e. g., object-oriented) seem to be a sub-optimal choice for building these systems. Indeed, looking at the results obtained from Chapter 8, programming languages that can combine multiple paradigms (e. g., Python or Matlab) better fit the characteristics required by practitioners to build such systems. In addition, due to the restricted hardware dimension of mobile and IoT devices, programming languages need to be re-engineered to be more “memory-preserving”.

Moreover, big companies and open-source communities often opt for established programming languages like Java or Kotlin for developing IoT systems and mobile applications. However, these languages may not be well-suited for coding on low-level devices, leading to potential performance issues and decreased perceived software quality by end-users. Additionally, current programming languages typically operate within a single process, limiting software system performance or necessitating multi-process libraries. Unfortunately, employing such libraries may increase the likelihood of unexpected behaviors, particularly if developers lack experience with multiprocessing techniques.

Research Directions:

- The SE community must propose mechanisms, paradigms, and frameworks to facilitate the paradigm shift.
- Researchers need to focus on how programming languages can better support the development of systems that operate in agnostic contexts, such as IoT systems. This could involve exploring features of programming languages to seek greater abstraction to facilitate adaptation and ease the processes of maintenance and evolution.

Open Issues and Future Analyses

Despite the deeper investigation performed in this dissemination to understand Evolutionary Code Quality, a long way remains, and future analysis needs to be conducted to release context-dependent quality assurance tools. In this sense, due to the findings achieved in this dissertation, important future studies need to be done to comprehend how code quality evolves over time. More in detail, we noticed by the findings obtained from Chapter 6 that different programming languages can imply different code smells, and in this way, this could also imply that different families of software systems can have different code quality issues over time. Moreover, the introduction of AI-enabled systems into the market could not be overlooked, suggesting that also artificial intelligence components could suffer from the rise of code smells. In addition, a number of frameworks have been proposed to assess privacy and security concerns in systems that use emerging technologies. However, these frameworks are not applied in practice in big companies and open-source communities, and this could indicate a lack of awareness of the importance of preserving quality aspects such as privacy and security for these systems.

The following subsection provides some detail on AI-specific code smells and describes one of the most prominent frameworks to preserve privacy in systems that daily exchange and store sensitive information.

AI-Specific Code Smells

Table 11.1: List of AI-CSs Identified by Zang *et al.*.

Code Smell	Description	Pipeline Stage	Effect
Broadcasting Feature Not Used	This smell refers to when the developer uses a tensor without a broadcasting operation.	Model Training	Memory Issues
Chain Indexing	This smell refers to when a developer uses to access a single data of a data frame using "[][]".	Data Cleaning	Performance
Columns and DataType Not Explicitly Set	This smell refers to when a developer declares a data frame without declaring the column name and the data type.	Data Cleaning	Defect Proneness
Data Leakage	This smell refers to when the data used by developers to train the model contains prediction results information.	Model Evaluation	Defect Proneness
Data frame Conversion API Misused	This smell refers to when a developer uses the function .values() to transform a data frame object to a Numpy array.	Data Cleaning	Defect Proneness
Deterministic Algorithm Option Not Used	This smell refers to when a developer does not remove the option "deterministic_-algorithms(True)".	Model Training	Reproducibility
Empty Column Misinitialization	This smell refers to when a developer uses zeros values or empty strings to initialize a new column in a data frame.	Data Cleaning	Robustness
Gradients Not Cleared Before Backward Propagation	This smell refers to when a developer does not use "optimizer.zero_- grad()" before " loss_fn.backward()" to clear gradients.	Model Training	Defect Proneness
Hyperparameter Not Explicitly Set	This smell occurs when a developer does not explicitly set the hyperparameter of an AI algorithm.	Model Training	Defect Proneness & Reproducibility
In-Place APIs Misused	This smell refers to when the developer assumes the Pandas function returns an in-place value.	Data Cleaning	Defect Proneness
Matrix Multiplication API Misused	This smell refers to when the developer uses the function "np.dot" to multiply a Numpy matrix.	Data Cleaning	Readability
Memory Not Freed	This smell regards when a developer declares a machine learning model in a loop operation without using the function "clear_session()" at the end of the loop.	Model Training	Memory Issue
Merge API Parameter Not Explicitly Set	This smell refers to when a developer does not specify the options "How" and "On" during a Pandas merge operation.	Data Cleaning	Readability
Missing the Mask of Invalid Value	When a developer uses the function "tf.clip()"(or similar) in a deep learning model, the value of certain variables could vary unexpectedly, causing invalid operations.	Model Training	Defect Proneness
NaN Equivalence Comparison Misused	This smell refers to when a developer uses the function " np.nan" to compare a data frame value with a NaN value.	Data Cleaning	Defect Proneness
No Scaling before Scaling-Sensitive Operation	This smell occurs when a developer does not use the feature scale function after some sensitive operation.	Feature Engineering	Defect Proneness
Pytorch Call Method Misused	This smell regards when a developer uses to forward the input to the network the function. "self.net.forward()"	Model Training	Robustness
Randomness Uncontrolled	This smell occurs when a developer does not explicitly set the random seed in the training process.	Model Training & Model Evaluation	Reproducibility
TensorArray Not Used	If a developer initializes an array using the function "tf.constant" and assigns a value in a loop operation, it is necessary to use "tf.TensorArray()" avoiding possible errors.	Model Training	Efficiency & Defect Proneness
Threshold-Dependent Validation	This smell refers to when a developer combines dependent (e. g., F1-score) and independent (e. g., RUC) metrics to evaluate the performance of the machine learning model.	Model Evaluation	Robustness
Training / Evaluation Mode Improper Toggling	In deep learnin code, it is crucial to call the training mode at the correct location to prevent the oversight of forgetting to switch back to the inference mode after the training step.	Model Training	Defect Proneness
Unnecessary Iteration	This smell regards when a developer uses a loop operation rather than the corresponding Pandas function.	Data Cleaning	Efficiency

The state-of-the-art in the context of AI-enabled Systems underlines a lack of knowledge of the empirical investigation of code smells that can arise during the building of AI-enabled Systems. In this respect, code smell identification still represents one of the most challenging tasks for a software engineer who builds complex software systems daily. The increased focus of the software engineering community on code smells mixed with the increased adoption of AI worldwide opened the door to identifying possible "*AI-specifics code smells*" i. e., code smells that can appear in the AI pipeline. Zhang *et al.* [395] recently released a catalog of 22 AI-CSs by empirically analyzing white and grey literature.

Table 11.1 shows the AI-CSs identified by the authors, their description, the pipeline stage they affect, and the quality aspects they impact.

To provide a tangible example of AI-CS, let consider *Gradients Not Cleared before Backward Propagation*. It refers to when a developer builds a neural network in a loop operation and does not use the function `optimizer.zero_grad()` to clear the old gradients at the end of each iteration. Without this operation, the gradients will gather from all the preceding backward calls. This situation can lead to a gradient explosion, causing a failure in the training process [363]. To mitigate this smell, the function `optimizer.zero_grad()` should be used before the backpropagation step. Listing 11.1 shows an example of *Gradients Not Cleared before Backward Propagation* smell for the project TRANSFORMERS.[1] We added an extra line (in green) to indicate how to refactor the smell in the taxonomy of Zhang *et al.* [395].

[1]https://github.com/huggingface/transformers/blob/main/examples/research_projects/bertology/run_bertology.py

```
for step, inputs in enumerate(tqdm(eval_dataloader,
  desc="Iteration", disable=args.local_rank not in [-1,
  0])):
   for k, v in inputs.items():
         optimizer.zero_grad()
        inputs[k] = v.to(args.device)
   outputs = model(**inputs, head_mask=head_mask)
   loss, logits, all_attentions = (
           outputs[0],
           outputs[1],
           outputs[-1],
       )
   loss.backward()  # Back propagate to populate the
  gradients in the head mask
```

Listing 11.1: Example of Gradients Not Cleared before Backward Propagation in the Transformer project.

Privacy by Design: Challenges and New Opportunities

The continuous generation of sensitive data provided by AI-enabled Systems (as discussed in previous chapters) quickly changes the perspective of how user data needs to be exchanged, generating important concerns about quality aspects that can, in turn, encourage users to abandon the software system prematurely due to lack of reliability. In this respect, a promising approach to mitigate possible premature software dismissal due to privacy concerns is the so-called "`Privacy by Design`" (PbD) framework. The term refers to proactive techniques for embedding privacy-preserving mechanisms in the systems lifecycle (i. e., from requirement elicitation to maintenance) [58].

The framework establishes seven fundamental principles:[2]

[2] https://www.iso.org/standard/84977.html

PROACTIVE, NOT REACTIVE; PREVENTATIVE NOT REMEDIAL. This principle refers to preventing privacy issues in software systems before they appear. Proactive procedures aimed to prevent data breaches include: I) high-security level standards, II) the demonstrability of these high-security standards at all the stakeholders involved, and III) analyzing the current internal company policies to identify possible privacy weaknesses in protocols.

PRIVACY BY DEFAULT. Software systems that use sensitive data must ensure high privacy levels by embedding privacy mechanisms as default settings without asking users to enable specific privacy-preserving settings. This category includes: I) Collection limitation: the software system must collect data only using legal and clear procedures; II) Data minimization: The software system must collect only the minimum amount of data required to guarantee a correct system workflow; III) Use, retention & disclosure limitation: The collected data must be used only for the purpose than to which the user has agreed. Data that are no longer needed must be deleted; IV) Security: The data exchange must be only made using secure encryption protocols.

PRIVACY EMBEDDED INTO DESIGN. Data and privacy protection in websites, mobile apps, or software development is emphasized. It's not just an add-on feature but should be embedded seamlessly throughout the design process. This requires a privacy-first mindset where every decision prioritizes functionality and safeguarding user privacy.

FULL FUNCTIONALITY – POSITIVE-SUM, NOT ZERO–SUM. A fatalistic outlook is incompatible with the concept of Privacy by Design. Those who believe trade-offs between user experience or security protocols are necessary to adopt a zero-sum mentality. Conversely, those who seamlessly integrate privacy into all design aspects take a positive-sum approach. These innovators are likely to witness their brands flourish in a landscape where privacy is not merely a legal requirement but also a significant factor influencing market dynamics.

END-TO-END SECURITY – LIFECYCLE PROTECTION. Privacy by Design guarantees the security of personal data throughout its entire lifecycle, from collection to destruction after serving its intended purpose. This comprehensive protection underscores the interdisciplinary nature of Privacy by Design, which relies on security best practices to ensure end-to-end data security. By upholding confidentiality, data integrity, and accessibility throughout its duration with the company, Privacy by Design ensures robust data protection at every processing stage.

VISIBILITY AND TRANSPARENCY – KEEP IT OPEN. Openly communicating privacy policies and procedures to users fosters accountability and trust. Privacy by Design entails documenting and communicating actions clearly, consistently, and transparently. It reflects a collective commitment to privacy as a responsibility, underscored by the team's dedication. This commitment should be reinforced by an accessible and efficient process for users to submit and resolve complaints, along with independent verification of policies and commitments to users.

RESPECT FOR USER PRIVACY – KEEP IT USER-CENTRIC. Respecting user privacy entails prioritizing their privacy interests and implementing safeguards and features to protect them. This respect guides every design decision, recognizing that the optimal user experience prioritizes privacy. It involves empowering users to manage their data and actively involving them in the process, thereby putting control in their hands.

To concretize the Privacy by Design concept, the software engineering community proposes a number of reusability design solutions focused on privacy coined the definition of "`Privacy Patterns`". The idea behind privacy Patterns is to provide a catalog of possible reusability solutions to preserve privacy in all the activities that involve stakeholders.[3] In this respect, Lenhard *et al.* [194] in 2017 conducted a literature review on the adoption of privacy patterns in software systems. Their results showed that although Privacy by Design is a legal requirement, there is a lack of evidence on the use of privacy patterns in practice. This lack of evidence can

[3] https://privacypatterns.org/

be motivated by multiple considerations: I) Privacy patterns could require knowledge about data treatments that are outside the normal competencies of a software engineer; II) The applicability of these patterns could be too elaborate, demotivating developers to implement them; and III) Th lack of awareness of developers about the existence of these standards.

In this respect, a more concrete investigation of privacy patterns is required to comprehend the motivations that do not permit developers to embed privacy patterns in software systems.

12 CONCLUSION

Despite the effort spent by the software engineering community on code quality and the elevated number of high-quality papers, there are still unexplored aspects and important limitations that allow difficult the definition of *Evolutionary Code Quality*. While the body of knowledge on software systems in the context of code quality is largely explored, previous studies suffer from limitations related to I) a lack of investigation of how code quality varies over time and II) the impact of reusability mechanisms on code quality, and how code quality evolve in systems other than Java. We performed four empirical investigations to address these limitations by considering the most adopted reusability mechanisms i. e., , the programming abstraction and design patterns, and an additional to comprehend how code quality evolves in AI-enabled Systems. In the context of programming abstraction (i. e., inheritance and delegation), we conducted two empirical experiments. In the first one, we investigated their relationship with code smells by selecting three well-known Java projects (i. e., JHotDraw, JEdit, and Apache Ant). Statistically, we assessed how programming abstractions are related to the emergence of code smells from an evolutionary standpoint. The main results of this experiment indicated that, on the one hand, programming abstractions increase over time, not statistically significant way. On the other hand, their adoption is often positively related to the decrease of code smell severity (see Chapter 3 for more details). Still, from the programming abstractions standpoint, we empirically investigated how these mechanisms vary over time and how they are related to the increase of defect proneness and effort to fix bugs. We selected 12 Java projects provided by Defects4J and over 44k commits to perform this. From the evolutionary standpoint, our results corroborated our previous work, showing that programming abstractions vary over time but without a specific pattern, i. e., the patterns seem to be project-dependent. Furthermore, we discovered that programming abstractions are statistically significant for the above-mentioned tasks (see Chapter 4). Changing the reusability mechanism, we also investigated how design patterns are related to the emergence of code smells. We selected 15 Java projects with at least one design pattern and over 500 releases. Our findings showed that classes that participate in design patterns are often affected by code smells, and sometimes, there is a positive statistical correlation between the presence of specific design pattern instances (e. g., Bridge, Singleton, and Template Method) and the arising of specific code smells (e. g., God Class). This study showed that specific design pattern instances should be better planned to avoid sub-optimal implementations that could stimulate the presence of specific code smells (see Chapter 5).

Lastly, we explored code smell variation and the motivations for developers to introduce code smells in AI-enabled Systems over time. Specifically, to conduct our experiment, we selected 200 AI-enabled systems provided by NICHE [372] and analyzed I) the smell frequencies, II) the smell density, III) the survival time, and IV) The activities most frequently lead developers to introduce code smells in AI-enabled systems. The findings of this work indicated that code smells Python-specific (e. g., Complex List Comprehension and Long Ternary Conditional) are the most frequent and long-lived smells and that the smell variation shows instability patterns of "increase-decrease" over time. Lastly, in the majority of cases, the introduction of smells is due to evolutionary activities (see Chapter 6).

We also investigated code quality attributes by focusing on security and privacy aspects in systems developed using emerging technologies, specifically, a Systematic Literature Review (SLR) and an empirical study were performed. We performed a Systematic Literature Review (see Chapter 8 to comprehend how artificial intelligence is related to privacy in IoT systems. Our main objectives were to investigate I) What the domains where IoT devices work are. II) The artificial intelligence techniques used by researchers to deal with privacy in IoT environments, III) What datasets and tasks did researchers investigate when considering privacy attributes are. Our findings indicated that most of the time, the primary studies do not specify the domain where these devices work, suggesting that their applicability could be domain-agnostic; the most frequent AI techniques applied to discover privacy issues is the Support Vector Machine (SVM), and often the datasets selected are not fitted to use in a real-world scenario (e. g., toy datasets), and, lastly, the more frequent tasks are related to network attacks and malware detection.

From the empirical viewpoint, we empirically compare three static analysis vulnerability tools for mobile apps. We selected over 6,500 real mobile apps, investigating the vulnerability types detectable by static analysis tools, the analyzability of mobile apps, the frequency of detection of vulnerabilities, and the complementary of static analysis tools to detect vulnerabilities. Our results indicated that the existing automatic static analysis tools only partially cover the OWASP top-10 of mobile vulnerability; most times, static analysis tools fail to detect vulnerability due to configuration errors. Tools can detect different vulnerabilities with different frequencies, and the best configuration to obtain an all-encompassing overview of vulnerability is by using a combination of multiple tools (see Chapter 9).

The empirical studies discussed in this thesis contribute to advancing the state-of-the-art in the context of code quality, taking into account code quality from an evolutionary standpoint.

Although the mentioned studies increase the state-of-the-art in code quality context, the topic needs future studies to comprehend what aspects practitioners should monitor to improve code quality attributes over time, and simultaneously, researchers need to investigate other unexplored domains to comprehend how code quality varies over time.

LIST OF PUBLICATIONS

The complete list of publications is reported below. The ∗ symbol highlights the publications discussed in this dissertation.

INTERNATIONAL JOURNAL PAPERS

J01 - **Giordano G.,** Ferrucci, F., and Palomba, F. (2022). On the Use of Artificial Intelligence to Deal with Privacy in IoT Systems: A Systematic Literature Review. Journal of Systems and Software (JSS), Vol. 193, 111475 ∗

J02 - Amato, F., Cicalese, M., Contrasto, L., Cubicciotti, G., D'Ambola, G., La Marca, A., Pagano, G., Tomeo, F., Robertazzi, G. A., Vassallo, G., Acampora, G., Vitiello, A., Catolino, G., **Giordano, G.,** Lambiase, S., Pontillo, V., Sellitto, G., Ferrucci, F., and Palomba, F. (2023). QuantuMoonLight: A low-code platform to experiment with quantum machine learning. SoftwareX, 22, 101399. ∗

J03 - **Giordano, G.,** Festa, G., Catolino, G., Palomba, F., Ferrucci, F., and Gravino, C., On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort. Empirical Software Engineering (EMSE) 29, (2024) ∗

INTERNATIONAL CONFERENCE PAPERS

C01 - **Giordano, G.,** Fasulo, A., Catolino, G., Palomba, F., Ferrucci, F., and Gravino, C. On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality. IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) ∗

C02 - **Giordano, G.,** Palomba, F., and Ferrucci, F. A Preliminary Conceptualization and Analysis on Automated Static Analysis Tools for Vulnerability Detection in Android App. 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) ∗

C03 - **Giordano, G.,** Pontillo, V., Annunziata, G., Cimino, A., Ferrucci, F., and Palomba, F. How May Deep Learning Testing Inform Model Generalizability? The Case of Image Classification. In Proceedings of the 15th Seminar on Advanced Techniques & Tools for Software Evolution (SATToSE)

C04 - **Giordano, G.,** Sellitto, G., Sepe, A., Palomba, F., and Ferrucci, F. The Yin and Yang of Software Quality: On the Relationship between Design Patterns and Code Smells. 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) ∗

C05 - **Giordano, G.,** Annunziata, De Lucia, A., and Palomba, F. Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Software: A Preliminary Study. International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement 2023 ∗

BIBLIOGRAPHY

[1] Mousa AL-Akhras, Mohammed Alawairdhi, Ali Alkoudari, and Samer Atawneh. “Using Machine Learning to Build a Classification Model for IoT Networks to Detect Attack Signatures.” In: *International journal of Computer Networks & Communications* 12 (Nov. 2020), pp. 99–116. DOI: 10.5121/ijcnc.2020.12607.

[2] F Brito e Abreu and Walcelio Melo. “Evaluating the impact of object-oriented design on software quality.” In: *Proceedings of the 3rd international software metrics symposium*. IEEE. 1996, pp. 90–99.

[3] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and Selcuk Uluagac. “Peek-a-boo: I see your smart home activities, even encrypted!” In: *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 2020, pp. 207–218.

[4] Firas Shihab Ahmed, Norwati Mustapha, Aida Mustapha, Mohsen Kakavand, and Cik Feresa Mohd Foozy. “Preliminary Analysis of Malware Detection in Opcode Sequences within IoT Environment.” In: *Journal of Computer Science* 16.9 (2020), pp. 1306–1318.

[5] Yaser Al Mtawa, Harsimranjit Singh, Anwar Haque, and Ahmed Refaey. “Smart Home Networks: Security Perspective and ML-based DDoS Detection.” In: *2020 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE. 2020, pp. 1–8.

[6] Mohammed S Alam and Son T Vuong. “Random forest classification for detecting android malware.” In: *2013 IEEE international conference on green computing and communications and IEEE Internet of Things and IEEE cyber, physical and social computing*. IEEE. 2013, pp. 663–669.

[7] Umar Albalawi. “A Comprehensive Analysis On Intrusion Detection In Iot Based Smart Environments Using Machine Learning Approaches.” In: *International Journal of Scientific & Technology Research* 9 (2020), pp. 1646–1652.

[8] Fawzi Albalooshi. “The Metrics of Multiple Inheritance and the Reusability of Code–Java and C++.” In: *Journal of Advances in Mathematics and Computer Science* (2016), pp. 1–12.

[9] Fawzi Albalooshi and Amjad Mahmood. “A Comparative Study on the Effect of Multiple Inheritance Mechanism in Java, C++, and Python on Complexity and Reusability of Code.” In: *International Journal of Advanced Computer Science and Applications* 8.6 (2017), pp. 109–116.

[10] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. “Understanding of a convolutional neural network.” In: *2017 International Conference on Engineering and Technology (ICET)*. Ieee. 2017, pp. 1–6.

[11] Noura Aleisa and Karen Renaud. “Privacy of the Internet of Things: a systematic literature review (extended discussion).” In: *arXiv preprint arXiv:1611.03340* (2016).

[12] Paul Allison. “When can you safely ignore multicollinearity.” In: *Statistical horizons* 5.1 (2012), pp. 1–2.

[13] Alaa Omran Almagrabi and AK Bashir. “A Classification-based Privacy-Preserving Decision-Making for Secure Data Sharing in Internet of Things Assisted Applications.” In: *Digital Communications and Networks* (2021).

[14] Jonas S Almeida. “Predictive non-linear modeling of complex data by artificial neural networks.” In: *Current opinion in biotechnology* 13.1 (2002), pp. 72–76.

[15] Ahmed Alshehri, Jacob Granley, and Chuan Yue. “Attacking and protecting tunneled traffic of smart home devices.” In: *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*. 2020, pp. 259–270.

[16] Anas Alsoliman, Giulio Rigoni, Marco Levorato, Cristina Pinotti, Nils Ole Tippenhauer, and Mauro Conti. “COTS Drone Detection using Video Streaming Characteristics.” In: *International Conference on Distributed Computing and Networking 2021*. 2021, pp. 166–175.

[17] Badraddin Alturki, Stephan Reiff-Marganiec, and Charith Perera. “A hybrid approach for data analytics for internet of things.” In: *Proceedings of the Seventh International Conference on the Internet of Things*. 2017, pp. 1–8.

[18] Mohammed Amoon, Torki Altameem, and Ayman Altameem. “Internet of things sensor assisted security and quality analysis for health care data sets using artificial intelligent based heuristic health management system.” In: *Measurement* 161 (2020), p. 107861.

[19] Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Sofia Charalampidou, and Paris Avgeriou. “The effect of GoF design patterns on stability: a case study.” In: *IEEE Transactions on Software Engineering* 41.8 (2015), pp. 781–802.

[20] Chintan Amrit and Jos Van Hillegersberg. “Exploring the impact of soclo-technlcal core-periphery structures in open source software development.” In: *journal of information technology* 25.2 (2010), pp. 216–229.

[21] *An extensible cross-language static code analyzer.* `https://pmd.github.io/`.

[22] Ni An, Alexander Duff, Gaurav Naik, Michalis Faloutsos, Steven Weber, and Spiros Mancoridis. “Behavioral anomaly detection of malware on home routers.” In: *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE. 2017, pp. 47–54.

[23] Oscar Ancán and Carlos Cares. “Are Relevant the Code Smells on Maintainability Effort? A Laboratory Experiment.” In: *2018 IEEE International Conference on Automation/XXIII Congress of the Chilean*

Association of Automatic Control (ICA-ACCA). 2018, pp. 1–6. DOI: 10.1109/ICA-ACCA.2018.8609845.

[24] *Android ID Official Documentation*. https://developer.android.com/reference/android/provider/Settings.Secure.html#ANDROID_ID. Accessed: 2022-01-08.

[25] Eirini Anthi, Lowri Williams, Małgorzata Słowińska, George Theodorakopoulos, and Pete Burnap. "A supervised intrusion detection system for smart home IoT devices." In: *IEEE Internet of Things Journal* 6.5 (2019), pp. 9042–9053.

[26] Simon Duque Anton, Suneetha Kanoor, Daniel Fraunholz, and Hans Dieter Schotten. "Evaluation of machine learning-based anomaly detection algorithms on an industrial modbus/tcp data set." In: *Proceedings of the 13th international conference on availability, reliability and security*. 2018, pp. 1–9.

[27] Mattia Antonini, Massimo Vecchio, Fabio Antonelli, Pietro Ducange, and Charith Perera. "Smart audio sensors in the internet of things edge for anomaly detection." In: *IEEE Access* 6 (2018), pp. 67594–67610.

[28] Muhammad Aqeel, Fahad Ali, Muhammad Waseem Iqbal, Toqir A Rana, Muhammad Arif, Md Rabiul Auwul, et al. "A review of security and privacy concerns in the internet of things (IoT)." In: *Journal of Sensors* 2022 (2022).

[29] Pathum Chamikara Mahawaga Arachchige, Peter Bertok, Ibrahim Khalil, Dongxi Liu, Seyit Camtepe, and Mohammed Atiquzzaman. "Local differential privacy for deep learning." In: *IEEE Internet of Things Journal* 7.7 (2019), pp. 5827–5842.

[30] Pathum Chamikara Mahawaga Arachchige, Peter Bertok, Ibrahim Khalil, Dongxi Liu, Seyit Camtepe, and Mohammed Atiquzzaman. "A trustworthy privacy preserving framework for machine learning in industrial iot systems." In: *IEEE Transactions on Industrial Informatics* 16.9 (2020), pp. 6092–6102.

[31] Sevgi Arca and Rattikorn Hewett. “Privacy Protection in Smart Health.” In: *Proceedings of the 11th International Conference on Advances in Information Technology*. 2020, pp. 1–8.

[32] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[33] Kevin Ashton et al. “That ‘internet of things’ thing.” In: *RFID journal* 22.7 (2009), pp. 97–114.

[34] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. “Machine learning techniques for code smell detection: A systematic literature review and meta-analysis.” In: *Information and Software Technology* 108 (2019), pp. 115–138.

[35] Amin Azmoodeh, Ali Dehghantanha, and Kim-Kwang Raymond Choo. “Robust malware detection for internet of (battlefield) things devices using deep eigenspace learning.” In: *IEEE transactions on sustainable computing* 4.1 (2018), pp. 88–95.

[36] Ankit Bansal and Sudipta Mahapatra. “A comparative analysis of machine learning techniques for botnet detection.” In: *Proceedings of the 10th International Conference on Security of Information and Networks*. 2017, pp. 91–98.

[37] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. “Automatically securing permission-based software by reducing the attack surface: An application to android.” In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2012, pp. 274–277.

[38] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. “A validation of object-oriented design metrics as quality indicators.” In: *IEEE Transactions on software engineering* 22.10 (1996), pp. 751–761.

[39] Gustavo EAPA Batista, Ronaldo C Prati, and Maria Carolina Monard. “A study of the behavior of several methods for balancing machine learning training data.” In: *ACM SIGKDD explorations newsletter* 6.1 (2004), pp. 20–29.

[40] Samera Batool, Ali Hassan, Nazar Abbas Saqib, and Muazzam A Khan Khattak. “Authentication of Remote IoT Users Based on Deeper Gait Analysis of Sensor Data.” In: *IEEE Access* 8 (2020), pp. 101784–101796.

[41] Gueltoum Bendiab, Konstantinos-Panagiotis Grammatikakis, Ioannis Koufos, Nicholas Kolokotronis, and Stavros Shiaeles. “Advanced metering infrastructures: Security risks and mitigation.” In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. 2020, pp. 1–8.

[42] Donna Bergmark, Paradee Phempoonpanich, and Shumin Zhao. “Scraping the ACM digital library.” In: *ACM SIGIR Forum*. Vol. 35. 2. ACM New York, NY, USA. 2001, pp. 1–7.

[43] Sourav Bhattacharya, Dionysis Manousakas, Alberto Gil CP Ramos, Stylianos I Venieris, Nicholas D Lane, and Cecilia Mascolo. “Countering acoustic adversarial attacks in microphone-equipped smart home devices.” In: *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4.2 (2020), pp. 1–24.

[44] James M Bieman and Josephine Xia Zhao. “Reuse through inheritance: A quantitative study of C++ software.” In: *ACM SIGSOFT Software Engineering Notes* 20.SI (1995), pp. 47–52.

[45] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. “Don’t touch my code! Examining the effects of ownership on software quality.” In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, pp. 4–14.

[46] Alex Blewitt, Alan Bundy, and Ian Stark. “Automatic verification of design patterns in Java.” In: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 2005, pp. 224–232.

[47] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. “Lessons from applying the systematic liter-

ature review process within the software engineering domain." In: *Journal of systems and software* 80.4 (2007), pp. 571–583.

[48] Chris Brown. "Digital nudges for encouraging developer actions." In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE. 2019, pp. 202–205.

[49] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java.* 3rd. USA: Prentice Hall, 2009. ISBN: 0136061257.

[50] Joseph Bugeja, Andreas Jacobsson, and Paul Davidsson. "Is your home becoming a spy? a data-centered analysis and classification of smart connected home systems." In: *Proceedings of the 10th International Conference on the Internet of Things.* 2020, pp. 1–8.

[51] *CWE-284: Improper Access Control.* https://cwe.mitre.org/data/definitions/284.html. Accessed: 2022-01-11.

[52] Jose-Luis Cabra, Diego Mendez, and Luis C Trujillo. "Wide machine learning algorithms evaluation applied to ECG authentication and gender recognition." In: *Proceedings of the 2018 2nd International Conference on Biometric Engineering and Applications.* 2018, pp. 58–64.

[53] Victor R Basili1 Gianluigi Caldiera and H Dieter Rombach. "The goal question metric approach." In: *Encyclopedia of software engineering* (1994), pp. 528–532.

[54] Hui Cao, Shubo Liu, Renfang Zhao, and Xingxing Xiong. "IFed: A novel federated learning framework for local differential privacy in Power Internet of Things." In: *International Journal of Distributed Sensor Networks* 16.5 (2020), p. 1550147720919698.

[55] Davide Caputo, Luca Verderame, Andrea Ranieri, Alessio Merlo, and Luca Caviglione. "Fine-hearing Google Home: why silence will not protect your privacy." In: *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.* 11.1 (2020), pp. 35–53.

[56] Gemma Catolino, Fabio Palomba, Francesca Arcelli Fontana, Andrea De Lucia, Andy Zaidman, and Filomena Ferrucci. “Improving change prediction models with code smell-related information.” In: *Empirical Software Engineering* 25.1 (2020), pp. 49–95.

[57] Gemma Catolino, Fabio Palomba, Damian Andrew Tamburri, and Alexander Serebrenik. “Understanding Community Smells Variability: A Statistical Approach.” In: *International Conference on Software Engineering: Software Engineering in Society*. 2021, 77–86.

[58] Ann Cavoukian. “Understanding How to Implement Privacy by Design, One Step at a Time.” In: *IEEE Consumer Electronics Magazine* 9.2 (2020), pp. 78–82. DOI: 10.1109/MCE.2019.2953739.

[59] Mariano Ceccato, Andrea Capiluppi, Paolo Falcarin, and Cornelia Boldyreff. “A large study on the effect of code obfuscation on the quality of java code.” In: *Empirical Software Engineering* 20 (2015), pp. 1486–1524.

[60] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Bart Coppens, Bjorn De Sutter, Paolo Falcarin, and Marco Torchiano. “How professional hackers understand protected code while performing attack tasks.” In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE. 2017, pp. 154–164.

[61] Urbi Chatterjee, Soumi Chatterjee, Debdeep Mukhopadhyay, and Rajat Subhra Chakraborty. “Machine learning assisted PUF calibration for trustworthy proof of sensor data in IoT.” In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 25.4 (2020), pp. 1–21.

[62] Jagmohan Chauhan, Jathushan Rajasegaran, Suranga Seneviratne, Archan Misra, Aruna Seneviratne, and Youngki Lee. “Performance characterization of deep learning models for breathing-based authentication on resource-constrained devices.” In: *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2.4 (2018), pp. 1–24.

[63] Sonia Chawla and Rajender Nath. “Evaluating inheritance and coupling metrics.” In: *International Journal of Engineering Trends and Technology (IJETT)* 4.7 (2013), pp. 2903–2908.

[64] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. “Detecting code smells in Python programs.” In: *2016 international conference on Software Analysis, Testing and Evolution (SATE)*. IEEE. 2016, pp. 18–23.

[65] Zhifei Chen, Lin Chen, Wanwangying Ma, Xiaoyu Zhou, Yuming Zhou, and Baowen Xu. “Understanding metric-based detectable smells in Python software: A comparative study.” In: *Information and Software Technology* 94 (2018), pp. 14–29.

[66] Zhifei Chen, Lin Chen, Wanwangying Ma, Xiaoyu Zhou, Yuming Zhou, and Baowen Xu. “Understanding metric-based detectable smells in Python software: A comparative study.” In: *Information and Software Technology* 94 (2018), pp. 14–29. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.09.011. URL: https://www.sciencedirect.com/science/article/pii/S0950584916301690.

[67] Arti Chhikara, R Chhillar, and Sujata Khatri. “Evaluating the impact of different types of inheritance on the object oriented software metrics.” In: *International Journal of Enterprise Computing and Business Systems* 1.2 (2011), pp. 1–7.

[68] Shyam R Chidamber and Chris F Kemerer. “A metrics suite for object oriented design.” In: *IEEE Transactions on software engineering* 20.6 (1994), pp. 476–493.

[69] Te-Chuan Chiu, Yuan-Yao Shih, Ai-Chun Pang, Chieh-Sheng Wang, Wei Weng, and Chun-Ting Chou. “Semisupervised Distributed Learning With Non-IID Data for AIoT Service Platform.” In: *IEEE Internet of Things Journal* 7.10 (2020), pp. 9266–9277.

[70] Wonyoung Choi, Jisu Kim, SangEun Lee, and Eunil Park. “Smart home and internet of things: A bibliometric study.” In: *Journal of Cleaner Production* 301 (2021), p. 126908.

[71] Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. “Effective representation of aliases and indirect memory operations in SSA form.” In: *International Conference on Compiler Construction.* Springer. 1996, pp. 253–267.

[72] Norman Cliff. “Dominance statistics: Ordinal analyses to answer ordinal questions.” In: *Psychological bulletin* 114.3 (1993), p. 494.

[73] W. J. Conover. *Practical Non parametric Statistics.* 3rd ed. Wiley India Pvt. Limited, 2006.

[74] Iain D Craig. “Inheritance and Delegation.” In: *Object-Oriented Programming Languages: Interpretation.* Springer, 2007, pp. 83–128.

[75] William Crawford and Jonathan Kaplan. *J2EE design patterns: patterns in the real world.* " O’Reilly Media, Inc.", 2003.

[76] Marco D’Ambros, Alberto Bacchelli, and Michele Lanza. “On the impact of design flaws on software defects.” In: *2010 10th International Conference on Quality Software.* IEEE. 2010, pp. 23–31.

[77] John Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. “The effect of inheritance on the maintainability of object-oriented software: an empirical study.” In: *Proceedings of International Conference on Software Maintenance.* IEEE. 1995, pp. 20–29.

[78] John Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. “Evaluating inheritance depth on the maintainability of object-oriented software.” In: *Empirical Software Engineering* 1.2 (1996), pp. 109–132.

[79] Hamid Darabian, Ali Dehghantanha, Sattar Hashemi, Sajad Homayoun, and Kim-Kwang Raymond Choo. “An opcode-based technique for polymorphic Internet of Things malware detection.” In: *Concurrency and Computation: Practice and Experience* 32.6 (2020), e5173.

[80] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. “Design pattern recovery through visual language parsing and source code analysis.” In: *Journal of Systems and Software* 82.7 (2009), pp. 1177–1193.

[81] Manuel De Stefano, Fabiano Pecorelli, Fabio Palomba, and Andrea De Lucia. “Comparing within-and cross-project machine learning algorithms for code smell detection.” In: *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution*. 2021, pp. 1–6.

[82] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. “Rico: A mobile app dataset for building data-driven design applications.” In: *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 2017, pp. 845–854.

[83] Anthony Desnos. “Android: Static Analysis Using Similarity Distance.” In: *2012 45th Hawaii International Conference on System Sciences*. 2012, pp. 5394–5403. DOI: 10.1109/HICSS.2012.114.

[84] Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. “A developer centered bug prediction model.” In: *IEEE Transactions on Software Engineering* 44.1 (2017), pp. 5–24.

[85] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. “Detecting code smells using machine learning techniques: are we there yet?” In: *International conference on software analysis, evolution and reengineering (SANER)*. IEEE. 2018, pp. 612–621.

[86] Fei Ding, Hongda Li, Feng Luo, Hongxin Hu, Long Cheng, Hai Xiao, and Rong Ge. “DeepPower: Non-intrusive and deep learning-based detection of IoT malware using power side channels.” In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020, pp. 33–46.

[87] Lisa Nguyen Quang Do, James Wright, and Karim Ali. “Why do software developers use static analysis tools? a user-centered study of developer needs and motivations.” In: *IEEE Transactions on Software Engineering* (2020).

[88] Shuaike Dong, Zhou Li, Di Tang, Jiongyi Chen, Menghan Sun, and Kehuan Zhang. “Your smart home can’t keep a secret: Towards automated fingerprinting of iot traffic.” In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020, pp. 47–59.

[89] Wei Du, Ang Li, Pan Zhou, Zichuan Xu, Xiumin Wang, Hao Jiang, and Dapeng Wu. “Approximate to Be Great: Communication Efficient and Privacy-Preserving Large-Scale Distributed Deep Learning in Internet of Things.” In: *IEEE Internet of Things Journal* 7.12 (2020), pp. 11678–11692.

[90] Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. “Automatic repair of real bugs: An experience report on the defects4j dataset.” In: (2015).

[91] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N Nguyen. “A large-scale empirical study of Java language feature usage.” In: (2013).

[92] Anas Abou El Kalam, Aissam Outchakoucht, and Hamza Es-Samaali. “Emergence-Based Access Control: New Approach to Secure the Internet of Things.” In: *Proceedings of the 1st International Conference on Digital Tools & Uses Congress*. 2018, pp. 1–11.

[93] Ahmed M Elmisery, Mirela Sertovic, and Brij B Gupta. “Cognitive privacy middleware for deep learning mashup in environmental IoT.” In: *IEEE Access* 6 (2017), pp. 8029–8041.

[94] Mohammed Faisal Elrawy, Ali Ismail Awad, and Hesham F. A. Hamed. “Intrusion detection systems for IoT-based smart environments: a survey.” In: *Journal of Cloud Computing* 7 (2018), pp. 1–20.

[95] Julian J Faraway. *Extending the linear model with R: generalized linear, mixed effects and nonparametric regression models.* Chapman and Hall/CRC, 2016.

[96] Farnaz Farid, Mahmoud Elkhodr, Fariza Sabrina, Farhad Ahamed, and Ergun Gide. "A smart biometric identity management framework for personalised IoT and cloud computing-based healthcare services." In: *Sensors* 21.2 (2021), p. 552.

[97] Jiaxuan Fei, Qigui Yao, Mingliang Chen, Xiangqun Wang, and Jie Fan. "The Abnormal Detection for Network Traffic of Power IoT Based on Device Portrait." In: *Scientific Programming* 2020 (2020).

[98] Wolfram Fenske, Sandro Schulze, Daniel Meyer, and Gunter Saake. "When code smells twice as much: Metric-based detection of variability-aware code smells." In: *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2015, pp. 171–180. DOI: 10.1109/SCAM.2015.7335413.

[99] Aidin Ferdowsi and Walid Saad. "Deep learning for signal authentication and security in massive internet-of-things systems." In: *IEEE Transactions on Communications* 67.2 (2018), pp. 1371–1387.

[100] Elizabeth Fife and Juan Orjuela. "The privacy calculus: Mobile apps and user perceptions of privacy and security." In: *International Journal of Engineering Business Management* 4.Godište 2012 (2012), pp. 4–11.

[101] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. "Identification and application of extract class refactorings in object-oriented systems." In: *Journal of Systems and Software* 85.10 (2012), pp. 2241–2260.

[102] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. "Automatic detection of bad smells in code: An experimental assessment." In: *J. Object Technol.* 11.2 (2012), pp. 5–1.

[103] Francesca Arcelli Fontana and Marco Zanoni. "Code smell severity classification using machine learning techniques." In: *Knowledge-Based Systems* 128 (2017), pp. 43–58.

[104] Martin Fowler. "Refactoring: Improving the design of existing code." In: *11th European Conference. Jyväskylä, Finland.* 1997.

[105] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[106] Jyoti Gajrani, Meenakshi Tripathi, Vijay Laxmi, Gaurav Somani, Akka Zemmari, and Manoj Singh Gaur. "Vulvet: Vetting of Vulnerabilities in Android Apps to Thwart Exploitation." In: *Digital Threats: Research and Practice* 1.2 (2020). ISSN: 2692-1626. DOI: 10.1145/3376121. URL: https://doi.org/10.1145/3376121.

[107] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. "Design patterns: Abstraction and reuse of object-oriented design." In: *European Conference on Object-Oriented Programming.* Springer. 1993, pp. 406–431.

[108] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. "Design patterns: Abstraction and reuse of object-oriented design." In: *European Conference on Object-Oriented Programming.* Springer. 1993, pp. 406–431.

[109] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. "Understanding the evolution of android app vulnerabilities." In: *IEEE Transactions on Reliability* (2019).

[110] Robin Gassais, Naser Ezzati-Jivan, Jose M Fernandez, Daniel Aloise, and Michel R Dagenais. "Multi-level host-based intrusion detection system for Internet of things." In: *Journal of Cloud Computing* 9.1 (2020), pp. 1–16.

[111] Mattias T Gebrie and Habtamu Abie. "Risk-based adaptive authentication for Internet of things in smart home eHealth." In: *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings.* 2017, pp. 102–108.

[112] Franz-Xaver Geiger and Ivano Malavolta. "Datasets of Android applications: a literature review." In: *arXiv preprint arXiv:1809.10069* (2018).

[113] Franz-Xaver Geiger, Ivano Malavolta, Luca Pascarella, Fabio Palomba, Dario Di Nucci, and Alberto Bacchelli. “A graph-based dataset of commit history of real-world android apps.” In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 30–33.

[114] Smita Ghaisas, Preethu Rose, Maya Daneva, Klaas Sikkel, and Roel J Wieringa. “Generalizing by similarity: Lessons learnt from industrial case studies.” In: *2013 1st International Workshop on Conducting Empirical Studies in Industry (CESI)*. IEEE. 2013, pp. 37–42.

[115] Ashish Ghosh, Debasrita Chakraborty, and Anwesha Law. “Artificial intelligence in Internet of things.” In: *CAAI Transactions on Intelligence Technology* 3.4 (2018), pp. 208–218.

[116] Emanuel Giger, Martin Pinzger, and Harald C. Gall. “Can we predict types of code changes? An empirical analysis.” In: *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. 2012, pp. 217–226. DOI: `10.1109/MSR.2012.6224284`.

[117] Giammaria Giordano, Giusy Annunziata, Andrea De Lucia, and Fabio Palomba. *Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Software: A Preliminary Study — Online Appendix*. `https://figshare.com/s/d7b26dc76bc5c7aa06c8`. 2023.

[118] Giammaria Giordano, Antonio Fasulo, Gemma Catolino, Fabio Palomba, Filomena Ferrucci, and Carmine Gravino. “On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality.” In: *IEEE Inter. Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2022, pp. 1–12.

[119] Giammaria Giordano, Gerardo Festa, Gemma Catolino, Fabio Palomba, Filomena Ferrucci, and Carmine Gravino. *Web Appendix of the paper*. `https://giammariagiordano.github.io/On_the_Adoption_and_Effects_of_Source_Code_Reuse_on_Defect_Proneness_and_Maintenance_Effort/`. Online.

[120] Giammaria Giordano, Gerardo Festa, Gemma Catolino, Fabio Palomba, Filomena Ferrucci, and Carmine Gravino. “On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort.” In: *arXiv preprint arXiv:2208.07471* (2022).

[121] Michael W Godfrey and Daniel M German. “On the evolution of Lehman’s Laws.” In: *Journal of Software: Evolution and Process* 26.7 (2014), pp. 613–619.

[122] Brij Mohan Goel and Pradeep Kumar Bhatia. “Analysis of reusability of object-oriented systems using object-oriented metrics.” In: *ACM SIGSOFT Software Engineering Notes* 38.4 (2013), pp. 1–5.

[123] Chao Gong, Fuhong Lin, Xiaowen Gong, and Yueming Lu. “Intelligent cooperative edge computing in internet of things.” In: *IEEE Internet of Things Journal* 7.10 (2020), pp. 9372–9382.

[124] Neil Zhenqiang Gong, Mathias Payer, Reza Moazzezi, and Mario Frank. “Forgery-resistant touch-based authentication on mobile devices.” In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 2016, pp. 499–510.

[125] Peter Grogono and Brian Shearing. “Concurrent software engineering: preparing for paradigm shift.” In: *Proceedings of the 2008 C3S2E Conference*. C3S2E ’08. Montreal, Quebec, Canada: Association for Computing Machinery, 2008, 99–108. ISBN: 9781605581019. DOI: 10.1145/1370256.1370270. URL: https://doi.org/10.1145/1370256.1370270.

[126] Tianbo Gu, Allaukik Abhishek, Hao Fu, Huanle Zhang, Debraj Basu, and Prasant Mohapatra. “Towards Learning-automation IoT Attack Detection through Reinforcement Learning.” In: *2020 IEEE 21st International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*. IEEE. 2020, pp. 88–97.

[127] Zhitao Guan, Zefang Lv, Xianwen Sun, Longfei Wu, Jun Wu, Xiaojiang Du, and Mohsen Guizani. “A differentially private big data nonparametric Bayesian clustering algorithm in smart grid.” In:

IEEE Transactions on Network Science and Engineering 7.4 (2020), pp. 2631–2641.

[128] Tibor Gyimóthy, Rudolf Ferenc, and Istvan Siket. "Empirical validation of object-oriented metrics on open source software for fault prediction." In: *IEEE Transactions on Software engineering* 31.10 (2005), pp. 897–910.

[129] Stephan Haller. "The things in the internet of things." In: *Poster at the (IoT 2010). Tokyo, Japan, November* 5.8 (2010), pp. 26–30.

[130] Hyo-Sik Ham, Hwan-Hee Kim, Myung-Sup Kim, and Mi-Jung Choi. "Linear SVM-based android malware detection for reliable IoT services." In: *Journal of Applied Mathematics* 2014 (2014).

[131] Ayyoob Hamza, Hassan Habibi Gharakheili, Theophilus A Benson, and Vijay Sivaraman. "Detecting volumetric attacks on lot devices via sdn-based monitoring of mud activity." In: *Proceedings of the 2019 ACM Symposium on SDN Research.* 2019, pp. 36–48.

[132] Jun Han, Shijia Pan, Manal Kumar Sinha, Hae Young Noh, Pei Zhang, and Patrick Tague. "Sensetribute: smart home occupant identification via fusion across on-object sensing devices." In: *Proceedings of the 4th ACM International Conference on Systems for Energy-Efficient Built Environments.* 2017, pp. 1–10.

[133] Jan Hannemann and Gregor Kiczales. "Design pattern implementation in Java and AspectJ." In: *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* 2002, pp. 161–173.

[134] Rakib Ul Haque, ASM Hasan, Qingshan Jiang, and Qiang Qu. "Privacy-preserving K-nearest neighbors training over blockchain-based encrypted health data." In: *Electronics* 9.12 (2020), p. 2096.

[135] Mahmudul Hasan, Md Milon Islam, Md Ishrak Islam Zarif, and MMA Hashem. "Attack and anomaly detection in IoT sensors in IoT sites using machine learning approaches." In: *Internet of Things* 7 (2019), p. 100059.

[136] Chathuranga Hasantha. “A Systematic Review of Code Smell Detection Approaches.” In: *Journal of Advancement in Software Engineering and Testing* 4.1 (2021).

[137] Ahmed E Hassan. “Predicting faults using the complexity of code changes.” In: *2009 IEEE 31st international conference on software engineering*. IEEE. 2009, pp. 78–88.

[138] Foyzul Hassan, Shaikh Mostafa, Edmund S.L. Lam, and Xiaoyin Wang. “Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges.” In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2017, pp. 38–47. DOI: 10.1109/ESEM.2017.11.

[139] Wan Haslina Hassan et al. “Current research on Internet of Things (IoT) security: A survey.” In: *Computer networks* 148 (2019), pp. 283–294.

[140] Jane Huffman Hayes, Sandip C Patel, and Liming Zhao. “A metrics-based software maintenance effort model.” In: *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings*. IEEE. 2004, pp. 254–258.

[141] Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. “An empirical study on software defect prediction with a simplified metric set.” In: *Information and Software Technology* 59 (2015), pp. 170–190.

[142] Xiaofan He, Richeng Jin, and Huaiyu Dai. “Deep PDS-learning for privacy-aware offloading in MEC-enabled IoT.” In: *IEEE Internet of Things Journal* 6.3 (2018), pp. 4547–4555.

[143] Kenedi Heather, Kunal K Shah, Krishna K Venkatasubramanian, Hang Cai, Ken Hoyme, Michael Seeberger, and Grace Wiechman. “A novel authentication biometric for pacemakers.” In: *Proceedings of the 2018 IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies*. 2018, pp. 81–87.

[144] István Hegedus and Márk Jelasity. “Distributed differentially private stochastic gradient descent: An empirical study.” In: *2016 24th Euromicro international conference on parallel, distributed, and network-based processing (PDP)*. IEEE. 2016, pp. 566–573.

[145] Péter Hegedűs, Dénes Bán, Rudolf Ferenc, and Tibor Gyimóthy. “Myth or reality? analyzing the effect of design patterns on software maintainability.” In: *ASEA and DRBC 2012, held in conjunction with GST 2012, Jeju Island, Korea, November 28-December 2, 2012.* Springer. 2012.

[146] Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck. “On the extent and nature of software reuse in open source java projects.” In: *Top Productivity through Software Reuse: 12th International Conference on Software Reuse, ICSR 2011, Pohang, South Korea, June 13-17, 2011. Proceedings 12*. Springer. 2011, pp. 207–222.

[147] Felienne Hermans and Efthimia Aivaloglou. “Do code smells hamper novice programming? A controlled experiment on Scratch programs.” In: *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 2016, pp. 1–10. DOI: `10.1109/ICPC.2016.7503706`.

[148] Kim Herzig, Sascha Just, and Andreas Zeller. “The impact of tangled code changes on defect prediction models.” In: *Empirical Software Engineering* 21.2 (2016), pp. 303–336.

[149] Ali HeydariGorji, Siavash Rezaei, Mahdi Torabzadehkashi, Hossein Bobarshad, Vladimir Alves, and Pai H Chou. “Hypertune: Dynamic hyperparameter tuning for efficient distribution of dnn training over heterogeneous systems.” In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE. 2020, pp. 1–8.

[150] *How Many Smartphones Are In The World?* `https://www.bankmycell.com/blog/how-many-phones-are-in-the-world`. Accessed: 2021-12-13.

[151] Yong Huang, Wei Wang, Hao Wang, Tao Jiang, and Qian Zhang. “Authenticating on-body IoT devices: An adversarial learning approach.” In: *IEEE Transactions on Wireless Communications* 19.8 (2020), pp. 5234–5245.

[152] Fatima Hussain, Rasheed Hussain, Syed Ali Hassan, and Ekram Hossain. “Machine learning in IoT security: Current solutions and future challenges.” In: *IEEE Communications Surveys & Tutorials* 22.3 (2020), pp. 1686–1721.

[153] Brian Huston. “The effects of design pattern application on metric scores.” In: *Journal of Systems and Software* 58.3 (2001), pp. 261–269.

[154] AKM Iqtidar Newaz, Amit Kumar Sikder, Mohammad Ashiqur Rahman, and A Selcuk Uluagac. “HealthGuard: A Machine Learning-Based Security Framework for Smart Healthcare Systems.” In: *arXiv e-prints* (2019), arXiv–1909.

[155] Md Shihabul Islam, Harsh Verma, Latifur Khan, and Murat Kantarcioglu. “Secure real-time heterogeneous iot data management system.” In: *2019 First IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE. 2019, pp. 228–235.

[156] Shivani Jain and Anju Saha. “Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection.” In: *Science of Computer Programming* 212 (2021), p. 102713.

[157] Jiajun Jiang, Yingfei Xiong, and Xin Xia. “A manual inspection of Defects4J bugs and its implications for automatic program repair.” In: *Sci. China Inf. Sci.* 62.10 (2019), 200102:1–200102:16.

[158] Linshan Jiang, Rui Tan, Xin Lou, and Guosheng Lin. “On lightweight privacy-preserving collaborative learning for internet-of-things objects.” In: *Proceedings of the International Conference on Internet of Things Design and Implementation.* 2019, pp. 70–81.

[159] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. “Why don’t software developers use static analysis tools to find bugs?” In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 672–681.

[160] Théo Jourdan, Antoine Boutet, Amine Bahi, and Carole Frindel. “Privacy-Preserving IoT framework for activity recognition in personal healthcare monitoring.” In: *ACM Transactions on Computing for Healthcare* 2.1 (2020), pp. 1–22.

[161] Marian Jureczko. “Significance of different software metrics in defect prediction.” In: *Software Engineering: An International Journal* 1.1 (2011), pp. 86–95.

[162] Sai Praveen Kadiyala, Manaar Alam, Yash Shrivastava, Sikhar Patranabis, Muhamed Fauzi Bin Abbas, Arnab Kumar Biswas, Debdeep Mukhopadhyay, and Thambipillai Srikanthan. “LAMBDA: Lightweight assessment of malware for emBeddeD architectures.” In: *ACM Transactions on Embedded Computing Systems (TECS)* 19.4 (2020), pp. 1–31.

[163] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. “A large-scale empirical study of just-in-time quality assurance.” In: *IEEE Transactions on Software Engineering* 39.6 (2012), pp. 757–773.

[164] Renuga Kanagavelu, Zengxiang Li, Juniarto Samsudin, Yechao Yang, Feng Yang, Rick Siow Mong Goh, Mervyn Cheah, Praewpiraya Wiwatphonthana, Khajonpong Akkarajitsakul, and Shangguang Wang. “Two-phase multi-party computation enabled privacy-preserving federated learning.” In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE. 2020, pp. 410–419.

[165] Olumide Kayode and Ali Saman Tosun. “Analysis of iot traffic using http proxy.” In: *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE. 2019, pp. 1–7.

[166] Hannes Kegel and Friedrich Steimann. "Systematically refactoring inheritance to delegation in Java." In: *Proceedings of the 30th international conference on Software engineering*. 2008, pp. 431–440.

[167] Sean Kennedy, Haipeng Li, Chenggang Wang, Hao Liu, Boyang Wang, and Wenhai Sun. "I can hear your alexa: Voice command fingerprinting on smart home speakers." In: *2019 IEEE Conference on Communications and Network Security (CNS)*. IEEE. 2019, pp. 232–240.

[168] Rafiullah Khan, Sarmad Ullah Khan, Rifaqat Zaheer, and Shahid Khan. "Future internet: the internet of things architecture, possible applications and key challenges." In: *2012 10th international conference on frontiers of information technology*. IEEE. 2012, pp. 257–260.

[169] Shivanjali Khare and Michael Totaro. "Ensemble Learning for Detecting Attacks and Anomalies in IoT Smart Home." In: *2020 3rd International Conference on Data Intelligence and Security (ICDIS)*. IEEE. 2020, pp. 56–63.

[170] Raffi Khatchadourian and Hidehiko Masuhara. "Proactive Empirical Assessment of New Language Feature Adoption via Automated Refactoring: The Case of Java 8 Default Methods." In: *The Art, Science, and Engineering of Programming* 2.3 (2018), pp. 6–1.

[171] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. "An Empirical Study on the Use and Misuse of Java 8 Streams." In: *FASE*. 2020, pp. 97–118.

[172] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "An exploratory study of the impact of antipatterns on class change-and fault-proneness." In: *Empirical Software Engineering* 17.3 (2012), pp. 243–275.

[173] Foutse Khomh and Yann-Gael Gueheneuc. "Do Design Patterns Impact Software Quality Positively?" In: *2008 12th European Con-*

ference on Software Maintenance and Reengineering. 2008, pp. 274–278. DOI: 10.1109/CSMR.2008.4493325.

[174] Barbara Kitchenham, O Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. "Systematic literature reviews in software engineering–a systematic literature review." In: *Information and software technology* 51.1 (2009), pp. 7–15.

[175] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. "Systematic literature reviews in software engineering – A systematic literature review." In: *Information and Software Technology* 51.1 (2009). Special Section - Most Cited Articles in 2002 and Regular Research Papers, pp. 7–15. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2008.09.009. URL: https://www.sciencedirect.com/science/article/pii/S0950584908001390.

[176] A Gunes Koru, Dongsong Zhang, and Hongfang Liu. "Modeling the effect of size on defect proneness for open-source software." In: *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE. 2007, pp. 10–10.

[177] Carel P Kruger and Gerhard P Hancke. "Benchmarking Internet of things devices." In: *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*. IEEE. 2014, pp. 611–616.

[178] Vasiliy Krundyshev. "Neural network approach to assessing cybersecurity risks in large-scale dynamic networks." In: *13th International Conference on Security of Information and Networks*. 2020, pp. 1–8.

[179] Keyur Kulkarni and Ahmad Y Javaid. "Open source android vulnerability detection tools: a survey." In: *arXiv preprint arXiv:1807.11840* (2018).

[180] Noble Kumari and Anju Saha. "Effect of refactoring on software quality." In: *Proc. Conf Softw. Main t.* 2014, pp. 37–46.

[181] Murat Kuzlu, Corinne Fair, and Ozgur Guler. “Role of artificial intelligence in the Internet of Things (IoT) cybersecurity.” In: *Discover Internet of things* 1.1 (2021), pp. 1–14.

[182] Stefano Lambiase, Gemma Catolino, Damian A Tamburri, Alexander Serebrenik, Fabio Palomba, and Filomena Ferrucci. “Good Fences Make Good Neighbours? On the Impact of Cultural and Geographical Dispersion on Community Smells.” In: *IEEE/ACM International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*. ACM. 2022, to appear.

[183] Beth M Lange and Thomas G Moher. “Some strategies of reuse in an object-oriented programming environment.” In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1989, pp. 69–73.

[184] Gierad Laput, Yang Zhang, and Chris Harrison. “Synthetic sensors: Towards general-purpose sensing.” In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 2017, pp. 3986–3999.

[185] Shahid Latif, Zhuo Zou, Zeba Idrees, and Jawad Ahmad. “A novel attack detection scheme for the industrial internet of things using a lightweight random neural network.” In: *IEEE Access* 8 (2020), pp. 89337–89350.

[186] Duc-Phong Le, Huasong Meng, Le Su, Sze Ling Yeo, and Vrizlynn Thing. “Biff: A blockchain-based iot forensics framework with identity privacy.” In: *TENCON 2018-2018 IEEE Region 10 Conference*. IEEE. 2018, pp. 2372–2377.

[187] Tam Le and Matt W Mutka. “CapChain: A privacy preserving access control framework based on blockchain for pervasive environments.” In: *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE. 2018, pp. 57–64.

[188] Ronald J Leach. “Software metrics and software maintenance.” In: *Journal of Software Maintenance: Research and Practice* 2.2 (1990), pp. 133–142.

[189] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. “Occlumency: Privacy-preserving remote deep-learning inference using sgx.” In: *The 25th Annual International Conference on Mobile Computing and Networking*. 2019, pp. 1–17.

[190] Wei-Han Lee and Ruby Lee. “Implicit sensor-based authentication of smartphone users with smartwatch.” In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. 2016, pp. 1–8.

[191] Yen-Ting Lee, Tao Ban, Tzu-Ling Wan, Shin-Ming Cheng, Ryoichi Isawa, Takeshi Takahashi, and Daisuke Inoue. “Cross platform iot-malware family classification based on printable strings.” In: *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE. 2020, pp. 775–784.

[192] Manny M Lehman. “Laws of software evolution revisited.” In: *European Workshop on Software Process Technology*. Springer. 1996, pp. 108–124.

[193] Tao Lei, Zhan Qin, Zhibo Wang, Qi Li, and Dengpan Ye. “EveDroid: Event-aware Android malware detection against model degrading for IoT devices.” In: *IEEE Internet of Things Journal* 6.4 (2019), pp. 6668–6680.

[194] Jörg Lenhard, Lothar Fritsch, and Sebastian Herold. “A Literature Study on Privacy Patterns Research.” In: *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2017, pp. 194–201. DOI: 10.1109/SEAA.2017.28.

[195] Fengjun Li, Bo Luo, and Peng Liu. “Secure and privacy-preserving information aggregation for smart grids.” In: *International Journal of Security and Networks* 6.1 (2011), pp. 28–39.

[196] Huaxin Li, Zheyu Xu, Haojin Zhu, Di Ma, Shuai Li, and Kai Xing. “Demographics inference through Wi-Fi network traffic analysis.”

In: *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE. 2016, pp. 1–9.

[197] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. "Static analysis of android apps: A systematic literature review." In: *Information and Software Technology* 88 (2017), pp. 67–95.

[198] Wen Li, Xiaoqin Fu, and Haipeng Cai. "Androct: Ten years of app call traces in android." In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE. 2021, pp. 570–574.

[199] Xiaopeng Li, Fengyao Yan, Fei Zuo, Qiang Zeng, and Lannan Luo. "Touch well before use: Intuitive and secure authentication for iot devices." In: *The 25th annual international conference on mobile computing and networking*. 2019, pp. 1–17.

[200] William Lidwell, Kritina Holden, and Jill Butler. *Universal principles of design, revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design*. Rockport Pub, 2010.

[201] Mary G Lieberman and John D Morris. "The precise effect of multicollinearity on classification prediction." In: *Multiple Linear Regression Viewpoints* 40.1 (2014), pp. 5–10.

[202] Elvis Ligu, Alexander Chatzigeorgiou, Theodore Chaikalis, and Nikolaos Ygeionomakis. "Identification of refused bequest code smells." In: *2013 IEEE International Conference on Software Maintenance*. IEEE. 2013, pp. 392–395.

[203] Hui Lin, Sahil Garg, Jia Hu, Xiaoding Wang, Md Jalil Piran, and M Shamim Hossain. "Privacy-enhanced data fusion for COVID-19 applications in intelligent Internet of medical Things." In: *IEEE Internet of Things Journal* (2020).

[204] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. "A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications." In: *IEEE internet of things journal* 4.5 (2017), pp. 1125–1142.

[205] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. "An empirical study on android-related vulnerabilities." In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, pp. 2–13.

[206] Barbara H Liskov and Jeannette M Wing. "A behavioral notion of subtyping." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.6 (1994), pp. 1811–1841.

[207] Chi Harold Liu, Qiuxia Lin, and Shilin Wen. "Blockchain-enabled data collection and sharing for industrial IoT with deep reinforcement learning." In: *IEEE Transactions on Industrial Informatics* 15.6 (2018), pp. 3516–3526.

[208] Xiaoyuan Liu, Hongwei Li, Guowen Xu, Sen Liu, Zhe Liu, and Rongxing Lu. "PADL: Privacy-aware and asynchronous deep learning for IoT applications." In: *IEEE Internet of Things Journal* 7.8 (2020), pp. 6955–6969.

[209] Yang Liu, Yilong Yang, Zhuo Ma, Ximeng Liu, Zhuzhu Wang, and Siqi Ma. "PE-HEALTH: Enabling Fully Encrypted CNN for Health Monitor with Optimized Communication." In: *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. IEEE. 2020, pp. 1–10.

[210] Salvatore Longo and Bin Cheng. "Privacy preserving crowd estimation for safer cities." In: *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*. 2015, pp. 1543–1550.

[211] Hongmin Lu, Yuming Zhou, Baowen Xu, Hareton Leung, and Lin Chen. "The ability of object-oriented metrics to predict change-

proneness: a meta-analysis." In: *Empirical software engineering* 17.3 (2012), pp. 200–242.

[212] Rongxing Lu, Xiaohui Liang, Xu Li, Xiaodong Lin, and Xuemin Shen. "EPPA: An efficient and privacy-preserving aggregation scheme for secure smart grid communications." In: *IEEE Transactions on Parallel and Distributed Systems* 23.9 (2012), pp. 1621–1631.

[213] Yunlong Lu, Xiaohong Huang, Yueyue Dai, Sabita Maharjan, and Yan Zhang. "Blockchain and Federated Learning for Privacy-Preserved Data Sharing in Industrial IoT." In: *IEEE Transactions on Industrial Informatics* 16.6 (2020), pp. 4177–4186. DOI: 10.1109/TII.2019.2942190.

[214] Yunlong Lu, Xiaohong Huang, Ke Zhang, Sabita Maharjan, and Yan Zhang. "Communication-efficient federated learning and permissioned blockchain for digital twin edge networks." In: *IEEE Internet of Things Journal* 8.4 (2020), pp. 2276–2288.

[215] Chen Luo and Anshumali Shrivastava. "Arrays of (locality-sensitive) count estimators (ace) anomaly detection on the edge." In: *Proceedings of the 2018 World Wide Web Conference*. 2018, pp. 1439–1448.

[216] Lingjuan Lyu, James C Bezdek, Jiong Jin, and Yang Yang. "FORESEEN: Towards differentially private deep inference for intelligent Internet of Things." In: *IEEE Journal on Selected Areas in Communications* 38.10 (2020), pp. 2418–2429.

[217] *M2: Insecure Data Storage*. https://owasp.org/www-project-mobile-top-10/2016-risks/m2-insecure-data-storage. Accessed: 2022-01-11.

[218] *M3: Insecure Communication*. https://owasp.org/www-project-mobile-top-10/2016-risks/m3-insecure-communication. Accessed: 2022-01-11.

[219] *M5: Insufficient Cryptography*. https://owasp.org/www-project-mobile-top-10/2016-risks/m5-insufficient-cryptography. Accessed: 2022-01-11.

[220] Xindi Ma, Junying Zhang, Jianfeng Ma, Qi Jiang, Sheng Gao, and Kang Xie. "Do Not Perturb Me: A Secure Byzantine-Robust Mechanism for Machine Learning in IoT." In: *2020 International Conference on Networking and Network Applications (NaNA)*. IEEE. 2020, pp. 348–354.

[221] Carolyn Mair, Gada Kadoda, Martin Lefley, Keith Phalp, Chris Schofield, Martin Shepperd, and Steve Webster. "An investigation of machine learning based prediction systems." In: *Journal of systems and software* 53.1 (2000), pp. 23–29.

[222] AKM Jahangir Majumder and Joshua Aaron Izaguirre. "A Smart IoT Security System for Smart-Home Using Motion Detection and Facial Recognition." In: *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE. 2020, pp. 1065–1071.

[223] Mohammad Malekzadeh, Richard G Clegg, and Hamed Haddadi. "Replacement autoencoder: A privacy-preserving algorithm for sensory data analysis." In: *2018 IEEE/acm third international conference on internet-of-things design and implementation (iotdi)*. IEEE. 2018, pp. 165–176.

[224] Bomin Mao, Yuichi Kawamoto, and Nei Kato. "AI-based joint optimization of QoS and security for 6G energy harvesting Internet of Things." In: *IEEE Internet of Things Journal* 7.8 (2020), pp. 7032–7042.

[225] C Marinescu, R Marinescu, PF Mihancea, D Ratiu, and R Wettel. "iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design.[C]." In: *IEEE International Conference on Software Maintenance-industrial & Tool Volume*. DBLP. 2005.

[226] Radu Marinescu. "Assessing technical debt by identifying design flaws in software systems." In: *IBM Journal of Research and Development* 56.5 (2012), pp. 9–1.

[227] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. “A survey of app store analysis for software engineering.” In: *IEEE transactions on software engineering* 43.9 (2016), pp. 817–847.

[228] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. “Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset.” In: *Empirical Software Engineering* 22.4 (2017), pp. 1936–1964.

[229] Luana Martins, Heitor Costa, and Ivan Machado. “On the diffusion of test smells and their relationship with test code quality of Java projects.” In: *Journal of Software: Evolution and Process* (2023), e2532.

[230] Pedro Martins, André B Reis, Paulo Salvador, and Susana Sargento. “Physical layer anomaly detection mechanisms in IoT networks.” In: *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2020, pp. 1–9.

[231] M Hammad Mazhar and Zubair Shafiq. “Characterizing smart home iot traffic in the wild.” In: *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE. 2020, pp. 203–215.

[232] Deirdre N McCloskey and Stephen T Ziliak. “The standard error of regressions.” In: *Journal of economic literature* 34.1 (1996), pp. 97–114.

[233] Jason M McGinthy, Lauren J Wong, and Alan J Michaels. “Groundwork for neural network-based specific emitter identification authentication for IoT.” In: *IEEE Internet of Things Journal* 6.4 (2019), pp. 6429–6440.

[234] Shane McIntosh, Bram Adams, Thanh HD Nguyen, Yasutaka Kamei, and Ahmed E Hassan. “An empirical study of build maintenance effort.” In: *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE. 2011, pp. 141–150.

[235] Patrick E McKnight and Julius Najab. “Mann-Whitney U Test.” In: *The Corsini encyclopedia of psychology* (2010), pp. 1–1.

[236] Francesca Meneghello, Matteo Calore, Daniel Zucchetto, Michele Polese, and Andrea Zanella. “IoT: Internet of threats? A survey of practical security vulnerabilities in real IoT devices.” In: *IEEE Internet of Things Journal* 6.5 (2019), pp. 8182–8201.

[237] Tom Mens and Serge Demeyer. “Future trends in software evolution metrics.” In: *Proceedings of the 4th international workshop on Principles of software evolution*. 2001, pp. 83–86.

[238] Christian Meurisch, Bekir Bayrak, and Max Mühlhäuser. “Privacy-preserving AI services through data decentralization.” In: *Proceedings of The Web Conference 2020*. 2020, pp. 190–200.

[239] Bertrand Meyer. “Reusability: The case for object-oriented design.” In: *IEEE software* 4.2 (1987), p. 50.

[240] Jed Mills, Jia Hu, and Geyong Min. “Communication-efficient federated learning for wireless edge intelligence in IoT.” In: *IEEE Internet of Things Journal* 7.7 (2019), pp. 5986–5994.

[241] Minghui Min, Xiaoyue Wan, Liang Xiao, Ye Chen, Minghua Xia, Di Wu, and Huaiyu Dai. “Learning-based privacy-aware offloading for healthcare IoT with energy harvesting.” In: *IEEE Internet of Things Journal* 6.3 (2018), pp. 4307–4316.

[242] Saman Mirza Abdullah, Bilal Ahmed, and Musa M Ameen. “A new taxonomy of mobile banking threats, attacks and user vulnerabilities.” In: *Eurasian Journal of Science and Engineering* 3.3 (2018), pp. 12–20.

[243] *Mobile operating systems’ market share worldwide from January 2012 to June 2021*. `https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/`. Accessed: 2021-12-13.

[244] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. "Decor: A method for the specification and detection of code and design smells." In: *IEEE Transactions on Software Engineering* 36.1 (2009), pp. 20–36.

[245] Esraa Mohamed. "The relation of artificial intelligence with internet of things: A survey." In: *Journal of Cybersecurity and Information Management* 1.1 (2020), pp. 30–24.

[246] Hawzhin Mohammed, Syed Rafay Hasan, and Falah Awwad. "Fusion-On-Field Security and Privacy Preservation for IoT Edge Devices: Concurrent Defense Against Multiple Types of Hardware Trojan Attacks." In: *IEEE Access* 8 (2020), pp. 36847–36862.

[247] Bhabendu Kumar Mohanta, Debasish Jena, Utkalika Satapathy, and Srikanta Patnaik. "Survey on IoT security: Challenges and solution using machine learning, artificial intelligence and blockchain technology." In: *Internet of Things* 11 (2020), p. 100227.

[248] Nizar Msadek, Ridha Soua, and Thomas Engel. "Iot device fingerprinting: Machine learning based encrypted traffic analysis." In: *2019 IEEE wireless communications and networking conference (WCNC)*. IEEE. 2019, pp. 1–8.

[249] Anand Mudgerikar, Puneet Sharma, and Elisa Bertino. "Edge-based intrusion detection for IoT devices." In: *ACM Transactions on Management Information Systems (TMIS)* 11.4 (2020), pp. 1–21.

[250] Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab. "Studying permission related issues in android wearable apps." In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2018, pp. 345–356.

[251] John C Munson and Sebastian G Elbaum. "Code churn: A measure for estimating the impact of code change." In: *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE. 1998, pp. 24–31.

[252] Nachiappan Nagappan and Thomas Ball. “Use of relative code churn measures to predict system defect density.” In: *International conference on Software engineering*. 2005, pp. 284–292.

[253] TR Gopalakrishnan Nair and R Selvarani. “Defect proneness estimation and feedback approach for software design quality improvement.” In: *Information and software technology* 54.3 (2012), pp. 274–285.

[254] Fatemeh Nargesian, Horst Samulowitz, Udayan Khurana, Elias B Khalil, and Deepak S Turaga. “Learning Feature Engineering for Classification.” In: *Ijcai*. 2017, pp. 2529–2535.

[255] Emal Nasseri, Steve Counsell, and M Shepperd. “An empirical study of evolution of inheritance in Java OSS.” In: *19th Australian Conference on Software Engineering (ASWEC 2008)*. IEEE. 2008, pp. 269–278.

[256] Christopher Neff, Matías Mendieta, Shrey Mohan, Mohammadreza Baharani, Samuel Rogers, and Hamed Tabkhi. “REVAMP2T: Real-Time Edge Video Analytics for Multicamera Privacy-Aware Pedestrian Tracking.” In: *IEEE Internet of Things Journal* 7.4 (2020), pp. 2591–2602. DOI: `10.1109/JIOT.2019.2954804`.

[257] TJ OConnor, Reham Mohamed, Markus Miettinen, William Enck, Bradley Reaves, and Ahmad-Reza Sadeghi. “HomeSnitch: behavior transparency and control for smart home IoT devices.” In: *Proceedings of the 12th conference on security and privacy in wireless and mobile networks*. 2019, pp. 128–138.

[258] Mark Mbock Ogonji, George Okeyo, and Joseph Muliaro Wafula. “A survey on privacy and security of Internet of Things.” In: *Computer Science Review* 38 (2020), p. 100312.

[259] *Online Appendix*. `https://figshare.com/s/5aba382667d406194aea`.

[260] Seyed Ali Osia, Ali Shahin Shamsabadi, Sina Sajadmanesh, Ali Taheri, Kleomenis Katevas, Hamid R Rabiee, Nicholas D Lane, and Hamed Haddadi. "A hybrid deep learning architecture for privacy-preserving mobile analytics." In: *IEEE Internet of Things Journal* 7.5 (2020), pp. 4505–4518.

[261] Abdulhafis Abdulazeez Osuwa, Esosa Blessing Ekhoragbon, and Lai Tian Fat. "Application of artificial intelligence in Internet of Things." In: *2017 9th international conference on computational intelligence and communication networks (CICN)*. IEEE. 2017, pp. 169–173.

[262] Robert M O'brien. "A caution regarding rules of thumb for variance inflation factors." In: *Quality & quantity* 41.5 (2007), pp. 673–690.

[263] Marc-Oliver Pahl and François-Xavier Aubet. "All eyes on you: Distributed Multi-Dimensional IoT microservice anomaly detection." In: *2018 14th International Conference on Network and Service Management (CNSM)*. IEEE. 2018, pp. 72–80.

[264] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. "On the Diffuseness and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation." In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, p. 482. ISBN: 9781450356381. DOI: 10.1145/3180155.3182532. URL: https://doi.org/10.1145/3180155.3182532.

[265] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation." In: *Empirical Software Engineering* 23.3 (2018), pp. 1188–1221.

[266] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. "Mining ver-

sion histories for detecting code smells.” In: *IEEE Transactions on Software Engineering* 41.5 (2014), pp. 462–489.

[267] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. “Lightweight detection of android-specific code smells: The adoctor project.” In: *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE. 2017, pp. 487–491.

[268] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. “On the impact of code smells on the energy consumption of mobile applications.” In: *Information and Software Technology* 105 (2019), pp. 43–55. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.08.004. URL: https://www.sciencedirect.com/science/article/pii/S0950584918301678.

[269] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. “Toward a smell-aware bug prediction model.” In: *IEEE Transactions on Software Engineering* 45.2 (2017), pp. 194–218.

[270] Junjie Pang, Yan Huang, Zhenzhen Xie, Qilong Han, and Zhipeng Cai. “Realizing the heterogeneity: A self-organized federated learning framework for iot.” In: *IEEE Internet of Things Journal* 8.5 (2020), pp. 3088–3098.

[271] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. “Fine-grained just-in-time defect prediction.” In: *Journal of Systems and Software* 150 (2019), pp. 22–36.

[272] J-F Patenaude, Ettore Merlo, Michel Dagenais, and Bruno Laguë. “Extending software quality assessment techniques to java systems.” In: *Proceedings Seventh International Workshop on Program Comprehension*. IEEE. 1999, pp. 49–56.

[273] S Patro and Kishore Kumar Sahu. “Normalization: A preprocessing stage.” In: *arXiv preprint arXiv:1503.06462* (2015).

[274] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. “A systematic literature review on bad smells—5 W’s: which, when, what, who, where.” In: *IEEE Transactions on Software Engineering* (2018).

[275] Fabiano Pecorelli, Dario Di Nucci, Coen De Roover, and Andrea De Lucia. “A large empirical assessment of the role of data balancing in machine-learning-based code smell detection.” In: *Journal of Systems and Software* 169 (2020), p. 110693. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110693. URL: https://www.sciencedirect.com/science/article/pii/S0164121220301448.

[276] Fabiano Pecorelli, Fabio Palomba, Foutse Khomh, and Andrea De Lucia. “Developer-driven code smell prioritization.” In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 220–231.

[277] Anjana Perera. “Using Defect Prediction to Improve the Bug Detection Capability of Search-Based Software Testing.” In: *IEEE/ACM Inter. Conf. on Automated Software Engineering (ASE)*. 2020, pp. 1170–1174.

[278] Tran Nghi Phu, Le Huy Hoang, Nguyen Ngoc Toan, Nguyen Dai Tho, and Nguyen Ngoc Binh. “Cfdvex: A novel feature extraction method for detecting cross-architecture iot malware.” In: *Proceedings of the Tenth International Symposium on Information and Communication Technology*. 2019, pp. 248–254.

[279] Antônio J Pinheiro, Paulo Freitas de Araujo-Filho, Jeandro de M Bezerra, and Divanilson R Campelo. “Adaptive Packet Padding Approach for Smart Home Networks: A Tradeoff Between Privacy and Performance.” In: *IEEE Internet of Things Journal* 8.5 (2020), pp. 3930–3938.

[280] Daryl Posnett, Raissa D’Souza, Premkumar Devanbu, and Vladimir Filkov. “Dual ecological measures of focus in software develop-

ment." In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 452–461.

[281] Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter Tichy. "A controlled experiment on inheritance depth as a cost factor for code maintenance." In: *Journal of Systems and Software* 65.2 (2003), pp. 115 –126. ISSN: 0164-1212. DOI: https://doi.org/10.1016/S0164-1212(02)00053-5. URL: http://www.sciencedirect.com/science/article/pii/S0164121202000535.

[282] Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter Tichy. "A controlled experiment on inheritance depth as a cost factor for code maintenance." In: *Journal of Systems and Software* 65.2 (2003), pp. 115–126.

[283] Raneem Qaddoura, Ala'M Al-Zoubi, Iman Almomani, and Hossam Faris. "A Multi-Stage Classification Approach for IoT Intrusion Detection Based on Clustering with Oversampling." In: *Applied Sciences* 11.7 (2021), p. 3022.

[284] Attia Qamar, Ahmad Karim, and Victor Chang. "Mobile malware attacks: Review, taxonomy & future directions." In: *Future Generation Computer Systems* 97 (2019), pp. 887–909.

[285] Xiaoye Qian, Huan Chen, Haotian Jiang, Justin Green, Haoyou Cheng, and Ming-Chun Huang. "Wearable computing with distributed deep learning hierarchy: a study of fall detection." In: *IEEE Sensors Journal* 20.16 (2020), pp. 9408–9416.

[286] Youyang Qu, Longxiang Gao, Tom H Luan, Yong Xiang, Shui Yu, Bai Li, and Gavin Zheng. "Decentralized privacy using blockchain-enabled federated learning in fog computing." In: *IEEE Internet of Things Journal* 7.6 (2020), pp. 5171–5183.

[287] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. "Software fault prediction metrics: A systematic literature review." In: *Information and software technology* 55.8 (2013), pp. 1397–1418.

[288] Foyzur Rahman and Premkumar Devanbu. “How, and why, process metrics are better.” In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 432–441.

[289] Mohamed Abdur Rahman, M Shamim Hossain, Mohammad Saiful Islam, Nabil A Alrajeh, and Ghulam Muhammad. “Secure and provenance enhanced Internet of health things framework: A blockchain managed federated learning approach.” In: *Ieee Access* 8 (2020), pp. 205071–205087.

[290] Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, et al. “Empirical standards for software engineering research.” In: *arXiv preprint arXiv:2010.03525* (2020).

[291] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. “A Large Scale Study of Programming Languages and Code Quality in Github.” In: FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, 155–165. ISBN: 9781450330565. DOI: 10.1145/2635868.2635922. URL: https://doi.org/10.1145/2635868.2635922.

[292] Jingjing Ren, Daniel J Dubois, David Choffnes, Anna Maria Mandalari, Roman Kolcun, and Hamed Haddadi. “Information exposure from consumer iot devices: A multidimensional, network-informed measurement approach.” In: *Proceedings of the Internet Measurement Conference*. 2019, pp. 267–279.

[293] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. “A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software.” In: *IEEE Transactions on Software Engineering* 43.6 (2016), pp. 492–530.

[294] Ahmed Saeed, Ali Ahmadinia, Abbas Javed, and Hadi Larijani. “Intelligent intrusion detection in low-power IoTs.” In: *ACM Transactions on Internet Technology (TOIT)* 16.4 (2016), pp. 1–25.

[295] Kshira Sagar Sahoo and Deepak Puthal. “SDN-assisted DDoS defense framework for the Internet of multimedia things.” In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 16.3s (2020), pp. 1–18.

[296] Pasquale Salza, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Filomena Ferrucci. “Third-party libraries in mobile apps.” In: *Empirical Software Engineering* 25.3 (2020), pp. 2341–2377.

[297] Aamod Sane and Roy Campbell. “Object-oriented state machines: Subclassing, composition, delegation, and genericity.” In: *ACM Sigplan Notices* 30.10 (1995), pp. 17–32.

[298] Oluwafemi A Sarumi, Adebayo O Adetunmbi, and Fadekemi A Adetoye. “Discovering computer networks intrusion using data analytics and machine intelligence.” In: *Scientific African* 9 (2020), e00500.

[299] Stefano Savazzi, Monica Nicoli, and Vittorio Rampa. “Federated learning with cooperating devices: A consensus approach for massive IoT networks.” In: *IEEE Internet of Things Journal* 7.5 (2020), pp. 4641–4654.

[300] Francesco Schiliro, Nour Moustafa, and Amin Beheshti. “Cognitive Privacy: AI-enabled Privacy using EEG Signals in the Internet of Things.” In: *2020 IEEE 6th International Conference on Dependability in Sensor, Cloud and Big Data Systems and Application (DependSys)*. IEEE. 2020, pp. 73–79.

[301] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. “Machine learning: The high interest credit card of technical debt.” In: (2014).

[302] Jochen Seemann and Jürgen Wolff von Gudenberg. “Pattern-based design recovery of Java software.” In: *ACM SIGSOFT Software Engineering Notes* 23.6 (1998), pp. 10–16.

[303] Seung-Hyun Seo, Aditi Gupta, Asmaa Mohamed Sallam, Elisa Bertino, and Kangbin Yim. “Detecting mobile malware threats to homeland security through static analysis.” In: *Journal of Network and Computer Applications* 38 (2014), pp. 43–53.

[304] Grégory Seront, Miguel Lopez, Valérie Paulus, and Naji Habra. “On the relationship between Cyclomatic Complexity and the Degree of Object Orientation.” In: *Proc. of QAOOSE Workshop, ECOOP, Glasgow*. 2005, pp. 109–117.

[305] Shahab Shamshirband and Anthony T Chronopoulos. “A new malware detection system using a high performance-ELM method.” In: *Proceedings of the 23rd international database applications & engineering symposium*. 2019, pp. 1–10.

[306] Arun Sharma, PS Grover, and Rajesh Kumar. “Reusability assessment for software components.” In: *ACM SIGSOFT Software Engineering Notes* 34.2 (2009), pp. 1–6.

[307] Meng Shen, Baoli Ma, Liehuang Zhu, Xiaojiang Du, and Ke Xu. “Secure phrase search for intelligent processing of encrypted data in cloud-based IoT.” In: *IEEE Internet of Things Journal* 6.2 (2018), pp. 1998–2008.

[308] Meng Shen, Xiangyun Tang, Liehuang Zhu, Xiaojiang Du, and Mohsen Guizani. “Privacy-preserving support vector machine training over blockchain-based encrypted IoT data in smart cities.” In: *IEEE Internet of Things Journal* 6.5 (2019), pp. 7702–7712.

[309] Meng Shen, Huan Wang, Bin Zhang, Liehuang Zhu, Ke Xu, Qi Li, and Xiaojiang Du. “Exploiting Unintended Property Leakage in Blockchain-Assisted Federated Learning for Intelligent Edge Computing.” In: *IEEE Internet of Things Journal* 8.4 (2020), pp. 2265–2275.

[310] Faysal Hossain Shezan, Hang Hu, Jiamin Wang, Gang Wang, and Yuan Tian. “Read between the lines: An empirical measurement of sensitive applications of voice personal assistant systems.” In: *Proceedings of The Web Conference 2020*. 2020, pp. 1006–1017.

[311] Cong Shi, Jian Liu, Hongbo Liu, and Yingying Chen. "Smart user authentication through actuation of daily activities leveraging WiFi-enabled IoT." In: *Proceedings of the 18th ACM International Symposium on Mobile Ad Hoc Networking and Computing*. 2017, pp. 1–10.

[312] Nija Shi and Ronald A Olsson. "Reverse engineering of design patterns from java source code." In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE. 2006, pp. 123–134.

[313] Emad Shihab, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. "Is lines of code a good measure of effort in effort-aware models?" In: *Information and Software Technology* 55.11 (2013), pp. 1981–1993. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2013.06.002. URL: https://www.sciencedirect.com/science/article/pii/S0950584913001316.

[314] Shachar Siboni, Vinay Sachidananda, Yair Meidan, Michael Bohadana, Yael Mathov, Suhas Bhairav, Asaf Shabtai, and Yuval Elovici. "Security testbed for Internet-of-Things devices." In: *IEEE transactions on reliability* 68.1 (2019), pp. 23–44.

[315] Ravi Pratap Singh, Mohd Javaid, Abid Haleem, and Rajiv Suman. "Internet of things (IoT) applications to fight against COVID-19 pandemic." In: *Diabetes & Metabolic Syndrome: Clinical Research & Reviews* 14.4 (2020), pp. 521–524.

[316] Sarbjeet Singh, Sukhvinder Singh, and Gurpreet Singh. "Reusability of the Software." In: *International journal of computer applications* 7.14 (2010), pp. 38–41.

[317] Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. "Empirical validation of object-oriented metrics for predicting fault proneness models." In: *Software quality journal* 18.1 (2010), pp. 3–35.

[318] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. "Quantifying the effect of code smells on mainte-

nance effort.” In: *IEEE Transactions on Software Engineering* 39.8 (2012), pp. 1144–1156.

[319] Dag I.K. Sjøberg, Aiko Yamashita, Bente C.D. Anda, Audris Mockus, and Tore Dybå. “Quantifying the Effect of Code Smells on Maintenance Effort.” In: *IEEE Transactions on Software Engineering* 39.8 (2013), pp. 1144–1156. DOI: 10.1109/TSE.2012.89.

[320] Monika Skowron, Artur Janicki, and Wojciech Mazurczyk. “Traffic fingerprinting attacks on internet of things using machine learning.” In: *IEEE Access* 8 (2020), pp. 20386–20400.

[321] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. “Dissection of a bug dataset: Anatomy of 395 patches from Defects4J.” In: *International Conference on Software Analysis, Evolution and Reengineering, SANER*. IEEE Computer Society, 2018, pp. 130–140.

[322] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. “Dissection of a bug dataset: Anatomy of 395 patches from defects4j.” In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 130–140.

[323] Zéphyrin Soh, Aiko Yamashita, Foutse Khomh, and Yann-Gaël Guéhéneuc. “Do Code Smells Impact the Effort of Different Maintenance Programming Activities?” In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. 2016, pp. 393–402. DOI: 10.1109/SANER.2016.103.

[324] Ian Sommerville. “Software engineering 9th Edition.” In: *ISBN-10* 137035152 (2011), p. 18.

[325] Yubo Song, Qiang Huang, Junjie Yang, Ming Fan, Aiqun Hu, and Yu Jiang. “IoT device fingerprinting for relieving pressure in the access control.” In: *Proceedings of the ACM Turing Celebration Conference-China*. 2019, pp. 1–8.

[326] Neelam Soundarajan and Stephen Fridella. “Inheritance: From code reuse to reasoning reuse.” In: *Proceedings. Fifth International Conference on Software Reuse (Cat. No. 98TB100203)*. IEEE. 1998, pp. 206–215.

[327] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. “Pydriller: Python framework for mining software repositories.” In: *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 908–911.

[328] David L Spooner et al. “The impact of inheritance on security in object-oriented database systems.” In: *DBSec*. Citeseer. 1988, pp. 141–150.

[329] Ragav Sridharan, Rajib Ranjan Maiti, and Nils Ole Tippenhauer. “WADAC: Privacy-preserving anomaly detection and attack classification on wireless traffic.” In: *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. 2018, pp. 51–62.

[330] Christoph Stach and Frank Steimle. “Recommender-based privacy requirements elicitation-EPICUREAN: an approach to simplify privacy settings in iot applications with respect to the GDPR.” In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 2019, pp. 1500–1507.

[331] Leon Strous, Suné von Solms, and André Zúquete. “Security and privacy of the internet of things.” In: *Computers & Security* 102 (2021), p. 102148.

[332] Giancarlo Succi, Witold Pedrycz, Snezana Djokic, Paolo Zuliani, and Barbara Russo. “An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite.” In: *Empirical Software Engineering* 10.1 (2005), pp. 81–104.

[333] Aliya Tabassum, Aiman Erbad, Amr Mohamed, and Mohsen Guizani. “Privacy-Preserving Distributed IDS Using Incremen-

tal Learning for IoT Health Systems." In: *IEEE Access* 9 (2021), pp. 14271–14283.

[334] Damian A Tamburri, Fabio Palomba, and Rick Kazman. "Success and Failure in Software Engineering: A Followup Systematic Literature Review." In: *IEEE Transactions on Engineering Management* (2020).

[335] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. "An Empirical Study of Refactorings and Technical Debt in Machine Learning Systems." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 238–250.

[336] Richard Taylor. "Interpretation of the correlation coefficient: a basic review." In: *Journal of diagnostic medical sonography* 6.1 (1990), pp. 35–39.

[337] Ewan Tempero, Hong Yul Yang, and James Noble. "What programmers do with inheritance in Java." In: *European Conference on Object-Oriented Programming*. Springer. 2013, pp. 577–601.

[338] Geethapriya Thamilarasu, Adedayo Odesile, and Andrew Hoang. "An Intrusion Detection System for Internet of Medical Things." In: *IEEE Access* 8 (2020), pp. 181560–181576.

[339] Henri Theil. "A Multinomial Extension of the Linear Logit Model." In: *International Economic Review* 10.3 (1969), pp. 251–259. ISSN: 00206598, 14682354. URL: http://www.jstor.org/stable/2525642 (visited on 03/10/2023).

[340] Henri Theil. "A multinomial extension of the linear logit model." In: *International economic review* 10.3 (1969), pp. 251–259.

[341] Adam Thierer and Andrea Castillo. "Projecting the growth and economic impact of the internet of things." In: *George Mason University, Mercatus Center, June* 15 (2015).

[342] Navod Neranjan Thilakarathne. “Security and privacy issues in iot environment.” In: *International Journal of Engineering and Management Research* 10 (2020).

[343] Chin-Wei Tien, Shang-Wen Chen, Tao Ban, and Sy-Yen Kuo. “Machine learning framework to analyze iot malware using elf and opcode features.” In: *Digital Threats: Research and Practice* 1.1 (2020), pp. 1–19.

[344] Parastou Tourani, Bram Adams, and Alexander Serebrenik. “Code of conduct in open source projects.” In: *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE. 2017, pp. 24–33.

[345] Bernardo Trevizan, Jorge Chamby-Diaz, Ana LC Bazzan, and Mariana Recamonde-Mendoza. “A comparative evaluation of aggregation methods for machine learning over vertically partitioned data.” In: *Expert systems with applications* 152 (2020), p. 113406.

[346] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. “Design Pattern Detection Using Similarity Scoring.” In: *IEEE Transactions on Software Engineering* 32.11 (2006), pp. 896–909. DOI: 10.1109/TSE.2006.112.

[347] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. “When and why your code starts to smell bad (and whether the smells go away).” In: *IEEE Transactions on Software Engineering* 43.11 (2017), pp. 1063–1088.

[348] Eva Van Emden and Leon Moonen. “Java quality assurance by detecting code smells.” In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE. 2002, pp. 97–106.

[349] Jilles Van Gurp and Jan Bosch. “Design erosion: problems and causes.” In: *Journal of systems and software* 61.2 (2002), pp. 105–119.

[350] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*. Vol. 620. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[351] Carmine Vassallo, Fabio Palomba, Alberto Bacchelli, and Harald C Gall. "Continuous code quality: are we (really) doing that?" In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 790–795.

[352] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. "How developers engage with static analysis tools in different contexts." In: *Empirical Software Engineering* 25.2 (2020), pp. 1419–1457.

[353] Natthida Vatanapakorn, Chitsutha Soomlek, and Pusadee Seresangtakul. "Python Code Smell Detection Using Machine Learning." In: *2022 26th International Computer Science and Engineering Conference (ICSEC)*. 2022, pp. 128–133. DOI: 10.1109/ICSEC56337.2022.10049330.

[354] Petar Veličković, Nicholas D Lane, Sourav Bhattacharya, Angela Chieh, Otmane Bellahsen, and Matthieu Vegreville. "Scaling health analytics to millions without compromising privacy using deep distributed behavior models." In: *Proceedings of the 11th EAI International Conference on Pervasive Computing Technologies for Healthcare*. 2017, pp. 92–100.

[355] Santiago A Vidal, Claudia Marcos, and J Andrés Díaz-Pace. "An approach to prioritize code smells for refactoring." In: *Automated Software Engineering* 23.3 (2016), pp. 501–532.

[356] Santiago Vidal, Hernan Vazquez, J Andres Diaz-Pace, Claudia Marcos, Alessandro Garcia, and Willian Oizumi. "JSpIRIT: a flexible tool for the analysis of code smells." In: *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE. 2015, pp. 1–6.

[357] A Vinobha, Chitra Babu, et al. "Evaluation of reusability in aspect oriented software using inheritance metrics." In: *2014 IEEE international conference on advanced communications, control and computing technologies*. IEEE. 2014, pp. 1715–1722.

[358] Marek Vokáč, Walter Tichy, Dag IK Sjøberg, Erik Arisholm, and Magne Aldrin. "A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment." In: *Empirical Software Engineering* 9 (2004), pp. 149–195.

[359] *Vulnerability Details: CVE-2017-13236*. `https://www.cvedetails.com/cve/CVE-2017-13236/`. Accessed: 2022-01-19.

[360] Nazar Waheed, Xiangjian He, Muhammad Ikram, Muhammad Usman, Saad Sajid Hashmi, and Muhammad Usman. "Security and privacy in IoT using machine learning and blockchain: Threats and countermeasures." In: *ACM Computing Surveys (CSUR)* 53.6 (2020), pp. 1–37.

[361] Bartosz Walter and Tarek Alkhaeir. "The relationship between design patterns and code smells: An exploratory study." In: *Information and Software Technology* 74 (2016), pp. 127–142.

[362] Chenggang Wang, Sean Kennedy, Haipeng Li, King Hudson, Gowtham Atluri, Xuetao Wei, Wenhai Sun, and Boyang Wang. "Fingerprinting encrypted voice traffic on smart speakers with deep learning." In: *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 2020, pp. 254–265.

[363] Gan Wang, Zan Wang, Junjie Chen, Xiang Chen, and Ming Yan. "An Empirical Study on Numerical Bugs in Deep Learning Programs." In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–5.

[364] Haoyu Wang, Yao Guo, Zihao Tang, Guangdong Bai, and Xiangqun Chen. "Reevaluating android permission gaps with static and dynamic analysis." In: *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2015, pp. 1–6.

[365] Haoyu Wang, Yuanchun Li, Yao Guo, Yuvraj Agarwal, and Jason I Hong. “Understanding the purpose of permission use in mobile apps.” In: *ACM Transactions on Information Systems (TOIS)* 35.4 (2017), pp. 1–40.

[366] Hongtao Wang, Ang Li, Bolin Shen, Yuyan Sun, and Hongmei Wang. “Federated Multi-View Spectral Clustering.” In: *IEEE Access* 8 (2020), pp. 202249–202259.

[367] Ji Wang, Jianguo Zhang, Weidong Bao, Xiaomin Zhu, Bokai Cao, and Philip S Yu. “Not just privacy: Improving performance of private deep learning in mobile cloud.” In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018, pp. 2407–2416.

[368] Junjue Wang, Brandon Amos, Anupam Das, Padmanabhan Pillai, Norman Sadeh, and Mahadev Satyanarayanan. “Enabling live video analytics with a scalable and privacy-aware framework.” In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 14.3s (2018), pp. 1–24.

[369] Wei Wang, Fatjon Seraj, Nirvana Meratnia, and Paul JM Havinga. “Privacy-aware environmental sound classification for indoor human activity recognition.” In: *Proceedings of the 12th ACM International Conference on PErvasive Technologies Related to Assistive Environments*. 2019, pp. 36–44.

[370] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. “An empirical study of usages, updates and risks of third-party libraries in java projects.” In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2020, pp. 35–45.

[371] *Web Appendix of the paper*. `https://drive.google.com/drive/folders/18UkAkO4DeM5WXLyG3syo4qbL-_4__3X9?usp=sharing`.

[372] Ratnadira Widyasari, Zhou Yang, Ferdian Thung, Sheng Qin Sim, Fiona Wee, Camellia Lok, Jack Phan, Haodi Qi, Constance Tan, Qijin Tay, et al. “NICHE: A Curated Dataset of Engineered Machine Learning Projects in Python.” In: *arXiv preprint arXiv:2303.06286* (2023).

[373] Roel Wieringa and Maya Daneva. “Six strategies for generalizing software engineering theories.” In: *Science of computer programming* 101 (2015), pp. 136–152.

[374] Claes Wohlin. “Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering.” In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE ’14. London, England, United Kingdom: Association for Computing Machinery, 2014. ISBN: 9781450324762. DOI: 10.1145/2601248.2601268. URL: https://doi.org/10.1145/2601248.2601268.

[375] Claes Wohlin. “Second-generation systematic literature studies using snowballing.” In: *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. 2016, pp. 1–6.

[376] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[377] Hong Wu, Lin Shi, Celia Chen, Qing Wang, and Barry Boehm. “Maintenance effort estimation for open source software: A systematic literature review.” In: *IEEE international conference on software maintenance and evolution (ICSME)*. 2016, pp. 32–43.

[378] Hui Wu, Haiting Han, Xiao Wang, and Shengli Sun. “Research on artificial intelligence enhancing internet of things security: A survey.” In: *Ieee Access* 8 (2020), pp. 153826–153848.

[379] Jinbo Xiong, Mingfeng Zhao, Md Zakirul Alam Bhuiyan, Lei Chen, and Youliang Tian. “An AI-enabled three-party game framework for guaranteed data privacy in mobile edge crowdsensing of IoT.”

In: *IEEE Transactions on Industrial Informatics* 17.2 (2019), pp. 922–933.

[380] Dixing Xu, Mengyao Zheng, Linshan Jiang, Chaojie Gu, Rui Tan, and Peng Cheng. “Lightweight and unobtrusive data obfuscation at IoT edge for remote inference.” In: *IEEE Internet of Things Journal* 7.10 (2020), pp. 9540–9551.

[381] Haohang Xu, Jin Li, Hongkai Xiong, and Hui Lu. “FedMax: Enabling a Highly-Efficient Federated Learning Framework.” In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE. 2020, pp. 426–434.

[382] Yayin Xu, Ying Zhou, Przemyslaw Sekula, and Lieyun Ding. “Machine learning in construction: From shallow to deep learning.” In: *Developments in the Built Environment* 6 (2021), p. 100045.

[383] Aiko Yamashita and Leon 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.

[384] Yushuang Yan, Qingqi Pei, and Hongning Li. “Privacy-Preserving Compressive Model for Enhanced Deep-Learning-Based Service Provision System in Edge Computing.” In: *IEEE Access* 7 (2019), pp. 92921–92937.

[385] Lei Yang and Fengjun Li. “Cloud-Assisted Privacy-Preserving Classification for IoT Applications.” In: *2018 IEEE Conference on Communications and Network Security (CNS)*. IEEE. 2018, pp. 1–9.

[386] Lina Yang, Haiyu Deng, and Xiaocui Dang. “Preference preserved privacy protection scheme for smart home network system based on information hiding.” In: *IEEE Access* 8 (2020), pp. 40767–40776.

[387] Bo Yin, Hao Yin, Yulei Wu, and Zexun Jiang. “FDC: A secure federated deep learning mechanism for data collaborations in the Internet of Things.” In: *IEEE Internet of Things Journal* 7.7 (2020), pp. 6348–6359.

[388] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. “A reflective architecture for an object-oriented distributed operating system.” In: *Proceedings of the 1989 European Conference an Object Oriented Programming*. 1989, pp. 89–106.

[389] Jian Yu, Bin Fu, Ao Cao, Zhenqian He, and Di Wu. “EdgeCNN: A hybrid architecture for agile learning of healthcare data from IoT devices.” In: *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2018, pp. 852–859.

[390] Ping Yu, Tarja Systa, and Hausi Muller. “Predicting fault-proneness using OO metrics. An industrial case study.” In: *European Conference on Software Maintenance and Reengineering*. IEEE. 2002, pp. 99–107.

[391] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. “Characterizing logging practices in open-source software.” In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 102–112.

[392] Asimina Zaimi, Apostolos Ampatzoglou, Noni Triantafyllidou, Alexander Chatzigeorgiou, Androklis Mavridis, Theodore Chaikalis, Ignatios Deligiannis, Panagiotis Sfetsos, and Ioannis Stamelos. “An empirical study on the reuse of third-party libraries in open-source software development.” In: *Balkan Conference on Informatics Conference*. 2015, pp. 1–8.

[393] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. “Automated third-party library detection for android applications: Are we there yet?” In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2020, pp. 919–930.

[394] Cheng Zhang and David Budgen. “What Do We Know about the Effectiveness of Software Design Patterns?” In: *IEEE Transactions on Software Engineering* 38.5 (2012), pp. 1213–1231. DOI: 10.1109/TSE.2011.79.

[395] Haiyin Zhang, Luís Cruz, and Arie Van Deursen. “Code smells for machine learning applications.” In: *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI.* 2022, pp. 217–228.

[396] Ke Zhang, Siu Ming Yiu, and Lucas Chi Kwong Hui. “A Light-Weight Crowdsourcing Aggregation in Privacy-Preserving Federated Learning System.” In: *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2020, pp. 1–8.

[397] Mengyuan Zhang, Jiming Chen, Shibo He, Lei Yang, Xiaowen Gong, and Junshan Zhang. “Privacy-preserving database assisted spectrum access for industrial Internet of Things: A distributed learning approach.” In: *IEEE Transactions on Industrial Electronics* 67.8 (2019), pp. 7094–7103.

[398] Xiaofang Zhang, Yida Zhou, and Can Zhu. “An empirical study of the impact of bad designs on defect proneness.” In: *2017 International Conference on Software Analysis, Testing and Evolution (SATE)*. IEEE. 2017, pp. 1–9.

[399] Xiaoyu Zhang, Xiaofeng Chen, Joseph K Liu, and Yang Xiang. “DeepPAR and DeepDPA: privacy preserving and asynchronous deep learning for industrial IoT.” In: *IEEE Transactions on Industrial Informatics* 16.3 (2019), pp. 2081–2090.

[400] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, and Liming Zhu. “Making Python code idiomatic by automatic refactoring non-idiomatic Python code with pythonic idioms.” In: New York, NY, USA: Association for Computing Machinery, 2022. ISBN: 9781450394130. DOI: 10.1145/3540250.3549143. URL: https://doi.org/10.1145/3540250.3549143.

[401] Shengchu Zhao, Wei Li, Tanveer Zia, and Albert Y Zomaya. “A dimension reduction model and classifier for anomaly-based intrusion detection in internet of things.” In: *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big*

Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech). IEEE. 2017, pp. 836–843.

[402] Yang Zhao, Jun Zhao, Linshan Jiang, Rui Tan, Dusit Niyato, Zengxiang Li, Lingjuan Lyu, and Yingbo Liu. "Privacy-preserving blockchain-based federated learning for IoT devices." In: *IEEE Internet of Things Journal* 8.3 (2020), pp. 1817–1829.

[403] Huadi Zheng, Haibo Hu, and Ziyang Han. "Preserving user privacy for machine learning: local differential privacy or federated machine learning?" In: *IEEE Intelligent Systems* 35.4 (2020), pp. 5–14.

[404] Chunyi Zhou, Anmin Fu, Shui Yu, Wei Yang, Huaqun Wang, and Yuqing Zhang. "Privacy-preserving federated learning in fog computing." In: *IEEE Internet of Things Journal* 7.11 (2020), pp. 10782–10793.

[405] Pan Zhou, Guohui Zhong, Menglan Hu, Ruixuan Li, Qiben Yan, Kun Wang, Shouling Ji, and Dapeng Wu. "Privacy-Preserving and Residential Context-Aware Online Learning for IoT-Enabled Energy Saving With Big Data Support in Smart Home Environment." In: *IEEE Internet of Things Journal* 6.5 (2019), pp. 7450–7468.

[406] Tianqi Zhou, Jian Shen, Sai Ji, Yongjun Ren, and Leiming Yan. "Secure and Intelligent Energy Data Management Scheme for Smart IoT Devices." In: *Wireless Communications and Mobile Computing* 2020 (2020).

[407] Wei Zhou, Yiying Li, Shuhui Chen, and Bo Ding. "Real-time data processing architecture for multi-robots based on differential federated learning." In: *2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE. 2018, pp. 462–471.

[408] Yuming Zhou, Hareton Leung, and Baowen Xu. "Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness." In: *IEEE Transactions on Software Engineering* 35.5 (2009), pp. 607–623.

[409] Liehuang Zhu, Xiangyun Tang, Meng Shen, Xiaojiang Du, and Mohsen Guizani. "Privacy-preserving DDoS attack detection using cross-domain traffic in software defined networks." In: *IEEE Journal on Selected Areas in Communications* 36.3 (2018), pp. 628–643.

[410] Jan Henrik Ziegeldorf, Oscar Garcia Morchon, and Klaus Wehrle. "Privacy in the Internet of Things: threats and challenges." In: *Security and Communication Networks* 7.12 (2014), pp. 2728–2742.

[411] omitted. *Web Appendix of the paper*. `https://figshare.com/s/7c046b69d8f7ea75c0c8`. Online.

[412] omitted. *Web Appendix of the paper*. `https://giammariagiordano.github.io/TheYinAndYangOfSoftwareQuality/`. Online.

La borsa di dottorato è stata cofinanziata con risorse del
Programma Operativo Nazionale Ricerca e Innovazione 2014-2020 (CCI 2014IT16M2OP005),
Fondo Sociale Europeo, Azione I.1 “Dottorati Innovativi con caratterizzazione Industriale”

UNIONE EUROPEA
Fondo Sociale Europeo

Ministero dell'Istruzione, dell'Università e della Ricerca

PON
RICERCA
E INNOVAZIONE
2014 - 2020