Understanding Developer Practices in Java-based Software Development

Samiran Mahmud

Submitted in fulfillment of the requirements of the degree
of
Doctor of Philosophy

2013

Abstract

Software engineering is an interdisciplinary approach to solving problems where a result to a particular problem is formulated in terms of a software system. The means for mapping a desired solution design to a working software system are programming languages that provide different and, in general, disjoint sets of programming features. It is that set of features which furnishes developers with a feasible approach to express their *intentions* while developing software systems. While a well-balanced set of programming language features can empower developers, a feature mismatch may impose constraints on their decision making.

The use of specific language features in a solution design, however, can be influenced by varying guidelines, expert opinions, and community principles. As such cues are meant to encourage good programming practices, and improve quality (e.g., maintainability) of resulting software artifacts, it is expected that developers would adhere to the recommendations they are provided with. But to what extent do developers comply with such recommendations?

Even though many advances have been made in the field of software engineering, our knowledge of developers practices (whether they comply with recommendations they are provided with, in particular, and how they use language feature, in general) remains somewhat sketchy. Though existing literature provides us with some insights in this regard, there are still unexplored facets that, when investigated, can result in valuable insights into developers practices. Such an endeavor can assist us in constructing a strong link between programming language features and their application in practice, and also in identifying the extent to which developers adhere to associated recommendations.

In this thesis, we attempt to close the above gap. For this purpose, we investigate how developers employ a set of Java programming language features. The Java language offers various means for managing state and behavior of an object including *fields*, *properties* and *inner classes*. While fields are involved in representing the state of objects, properties provide us with a mechanism for defining so-called *getters* and *setters* that allow for controlled access to an object's state. Inner classes yield a convenient method to decompose desired functionality within a class into *nested classes*. These utilities facilitate the implementation of functionalities, but from two different perspectives (managing state vs. encapsulating behavior) and for three different levels of granularity (field vs. method vs. class).

We investigate the use of the above language features in the context of associated recommendations (i.e., guidelines, expert opinions, and community principles). The key to our understanding of developer practices with respect to the selected language features here is an appreciation of their typical usage patterns in Java-based software systems. We construct descriptive models, describe observations, and discuss the lesson learnt. We do not attempt, however, to establish any particular style guide on how developers must use fields, properties, or inner classes, but rather show how developers *usually* employ those features in practice.

As developers practices are imprinted in the code they produce, we choose to study program code. In fact, we investigate the Qualitas Corpus, a collection of more than 100 object-oriented Java-based software systems. We define set of software metrics and extract them from the Qualitas Corpus. We analyze the collected metrics with different statistical analysis techniques (i.e., frequency distribution analysis and inequality analysis) to answer our research questions.

The findings of this thesis advance our current understanding of developer behavior and decisions with regard to the use of language features (i.e., adherence to associated recommendations in particular, and the use-cases of studied features in general). It assist us, for example, to inform developers and managers about the current state of software systems, to support language designers to construct better programming languages, and to enrich software engineering education by reflecting developers' practices.

I dedicate this thesis to my parents.

Acknowledgements

This thesis would not be what it is without the cooperation of numerous people. First of all, I would like to acknowledge my supervisor Dr. Markus Lumpe. Without his continuous inspiration, suggestion and cooperation, this research work would not have been completed. I am indebted to him. I also acknowledge the cooperation of my associate supervisors Dr. Jean-Guy Schneider and Dr. Rajesh Vasa. Thanks to Olga Goloshchapova too. The collaborative work that I was involved in with them provided me with the opportunity to learn many different things. Moreover, their valuable feedbacks, insightful comments, and suggestions were very useful to improve my research work.

I would like to specially thank my family members for supporting me always. I thank all my colleagues and friends. I also thank the Swinburne University of Technology. This research work could not be accomplished without the necessary support (e.g., financial and other necessary resources) from this university. The list of acknowledgements remains incomplete without Dr. Ewan Tempero who provided me the access to Qualitas Corpus. I gratefully acknowledge it. Finally, I would like to thank the almighty Allah who has given me the strength and wisdom to accomplish this research work.

Samiran Mahmud

Declaration

I declare that the work presented in this thesis contains no material that has been accepted for the award of any other degree or diploma. To the best of my knowledge, it contains no material previously published (except this declaration) or written by another person (except where due reference is made). My own investigation resulted in the authentic outcomes of this thesis.

Samiran Mahmud

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis focuses on understanding developer behaviors and decisions, as represented in the program code they write, in terms of the use of programming language features. We investigated the use of language features in the context of associated recommendations that are meant to support good programming practices. Our work is based on an empirical analysis of hundreds of open source Java-based software systems. The outcomes can inform developers and managers about the current state of a software system, and can assist language designers to construct better programming languages.

In this chapter, we introduce briefly the research context of this work in section 1.1. We then present the research approach adopted in this work in section 1.2 and the key contributions in section 1.3. We conclude this chapter by presenting, in section 1.4, an outline of the thesis.

1.1 Research Context and Objective

Software developers use programming languages to express the capabilities of software systems. There are, however, differences in the way a programming language yields its expressive power to developers. In particular, we distinguish between language features that a programming language must support and language features that make a programming language more versatile [138]. Moreover, available programming language abstractions can influence developer effectiveness and decision making. According to Scott [187], "*language features clearly have a huge impact on the programmers ability to write clear, concise, and maintainable code, especially for very large systems*". Thus, while a well-defined set of features can empower developers, an ill-defined feature set can confine their creativity and productivity.

While employing programming language features in solution design, developer design choices can be influenced not only by the available language features, but also by associated recommendations. These include expert advices (e.g., [105, 183]), design guidelines (e.g., [12]), coding standard and conventions (e.g., [4, 10]). While some of this advice is concerned with how one should approach solution design (e.g., *tell, don't ask* principle [109, 110], *law of demeter* [132] in object-oriented design), some others are involved in supporting certain language abstractions through conventions (e.g., Java's property mechanism is supported by code conventions [10]). These guidelines are meant to encourage good programming practices and improve the quality (e.g., readability, maintainability) of resulting software artifacts [78]. It is, therefore, expected that developers would adhere to the recommendations they are provided with. But to what extent do developers comply with such recommendations?

The software engineering literature provides us with many different studies on available programming languages features (e.g., [62, 72, 73, 93, 140, 145, 208, 209, 217]) that relate to their usage patterns, appropriateness, and perceived quality. However, these studies focus only on selected and generally disjoint sets of features and, hence, do not provide us with (i) an understanding of how developers actually comply with given recommendations with regards to the use of programming language features in software development, and (ii) an approach to explore developer behaviors in this context.

Understanding what developers do in practice can assist (i) researchers to comprehend and model developers practices, and thus allows to document and reason about common and unusual (if any) trends in developer design decisions, (ii) managers to utilize the resulting observations to initiate any potential regulatory action when necessary, (iii) language designers to refine existing (and design new) language features that better reflect developer intentions, and (iv) peer developers and software engineering students to learn about accepted practices in software development.

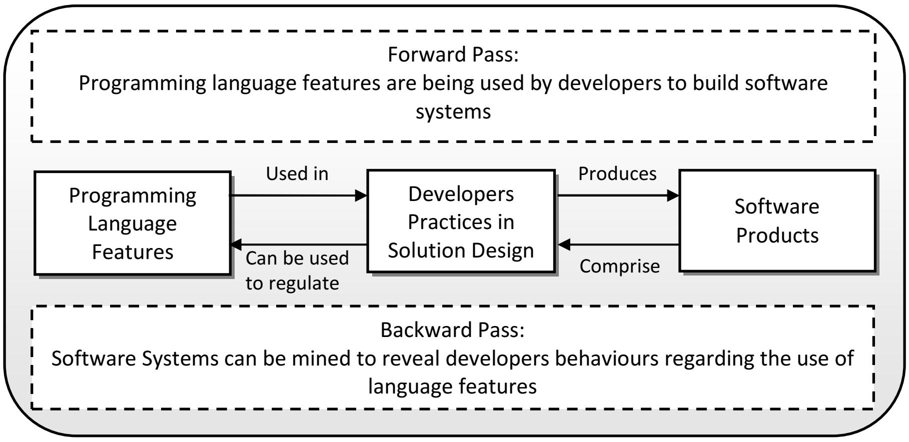

Forward Pass:
Programming language features are being used by developers to build software systems
Programming Language Features
Used in
Can be used to regulate
Developers Practices in Solution Design
Produces
Comprise
Software Products
Backward Pass:
Software Systems can be mined to reveal developers behaviours regarding the use of language features

Figure 1.1: Reflective Activity

An intriguing aspect of software is that the decisions and preferences of developers are imprinted in the program code they produce [217]. We can exploit this fact and mine existing products (i.e., software systems) in order to study how developers use specific programing language features in solution design [140]. We do not have to observe developers directly, but collect suitable metrics that would allow us to infer the developer design choices and preferences (cf. Figure 1.1).

In this thesis, we studied a set of features (i.e., fields, properties, and inner classes)[1] of the Java programming language in order to gain insights into, and build descriptive models of, developer behaviors in the context of associated recommendations. An overview of the research goals and outcomes are presented in Figure 1.2.

[1] Detail including rationale for selecting them is presented in chapter 2.

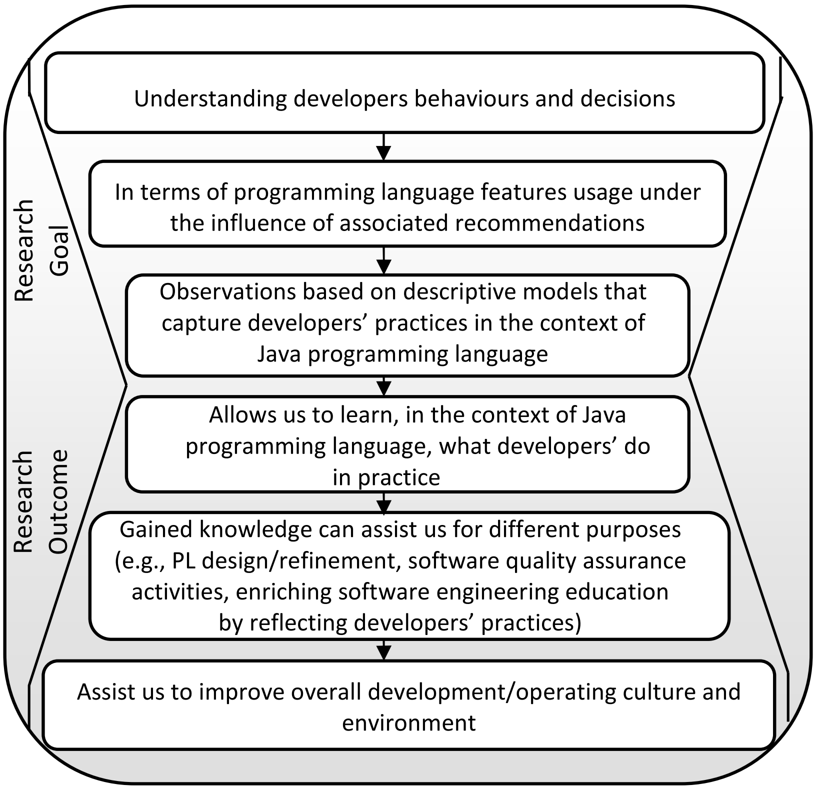

Understanding developers behaviours and decisions
In terms of programming language features usage under the influence of associated recommendations
Observations based on descriptive models that capture developers' practices in the context of Java programming language
Allows us to learn, in the context of Java programming language, what developers' do in practice
Gained knowledge can assist us for different purposes (e.g., PL design/refinement, software quality assurance activities, enriching software engineering education by reflecting developers' practices)
Assist us to improve overall development/operating culture and environment
Research Goal
Research Outcome

Figure 1.2: Research Context - Overview

1.2 Research Approach

We adopted an empirical research approach. Such approach, by its very nature, makes extensive use of quantitative information to yield a valid conclusion. As a source of such information, we used software artifacts (cf. Figure 1.3). In particular, we studied the Qualitas Corpus [77] that offers a collection of 106 open source Java-based software systems.

To extract the required information from the above collection of software artifacts, we used their compiled forms (i.e., Java class files comprising bytecode).[2] We defined set of software metrics and used a data mining framework [139] that distills Java class files to produce required metrics data. The collected metrics are used to study developer behaviors (as represented in the code they write) with regards to the use of programming language features under the influence of available coding

[2] Bytecode can be used as alternative to source code as evident by many studies (e.g., [35, 140, 144, 145, 206–209, 216]). Detail discussion is presented in chapter 3.

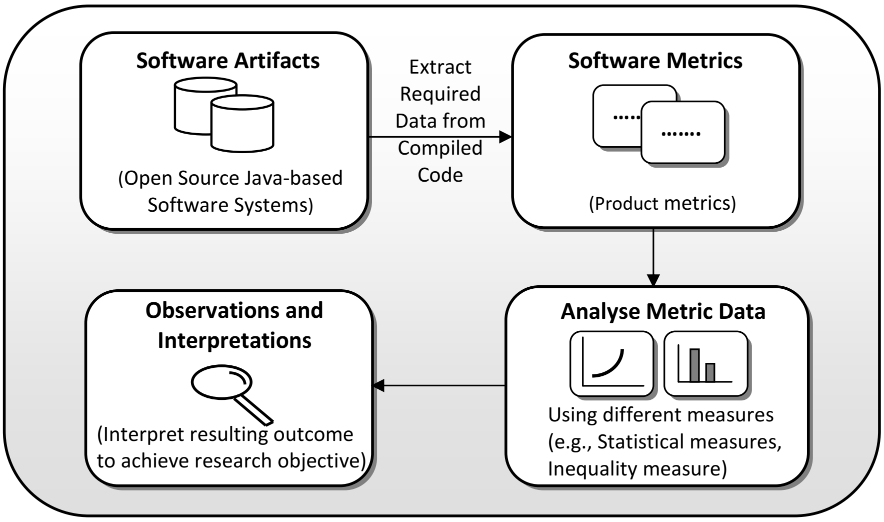

Software Artifacts
(Open Source Java-based Software Systems)
Extract Required Data from Compiled Code
Software Metrics
(Product metrics)
Analyse Metric Data
Using different measures (e.g., Statistical measures, Inequality measure)
Observations and Interpretations
(Interpret resulting outcome to achieve research objective)

Figure 1.3: Research Approach - Overview

standards, guidelines, and conventions. For example, we defined software metrics to capture the naming pattern of getter and setter methods in order to understand the developers' tendency in adhering to Java coding conventions [10].

The analysis of collected software metrics data is descriptive in nature. To analyze them, we employed different statistical measures and distribution-based analysis techniques. In particular, we used the Gini coefficient - a robust inequality-based measure to summarize software metrics data [216]. This approach has been used by Vasa [216] recently and found to be useful for software metrics data analysis because of the inherent nature of software metrics data (i.e., they follow skewed distribution profiles with long tails [35, 173, 228]). Furthermore, frequency distribution analysis of software metrics is a technique commonly used by many researchers (e.g., [49, 207–209, 225]).

Based on observations about aggregated metrics data, we model the typical patterns of developer behaviors in using language features under the influence of relevant recommendations they are provided with. For example, to understand whether developers circumvent visibility modifiers by using properties, we investigated the distribution (using the Gini coefficient) of, and also the correlation (using Spearman's rank correlation) between, private fields and getter setter methods in a given software system.

However, we do not attempted in this work to formulate any style guide on how developers *should* use language features in practice, rather we revealed what they *usually* do.

1.3 Research Outcomes

In this thesis, we confirm existing theories, as represented by recommendations in the literature (e.g., [12,69,105,106,114,183]), and present new insights into the developers practices with regards to the use of studied language features. The outcomes of this work are summarized below:

Developers tend to adhere to recommendations

We show that developers tend to follow advices, design guidelines, and code conventions offered to them. This is evident in case of the studied language features: fields, properties, and inner classes of Java programming language:

- In case of fields, advice like *all data should be hidden within its class* [183], *don't expose state if you don't have to* [12] are mostly followed with few exceptions. The extent of violations, however, is minimal. For example, when developers expose states (either deliberately or accidentally), they take advantages only in few cases.
- In case of properties, contrary to conventional belief, we show that they are neither commonplace nor evil. Given advices regarding whether to use or avoid getter and setter methods in literature (e.g., [69, 105, 106, 114, 183]), we show that developers proactively select getter and setter methods in order to satisfy specific domain requirements, not to circumvent data encapsulation.

- In case of inner classes, due to anonymous classes being perceived more clumsy and verbose as a consequence of their bulky syntax [184] and associated readability concerns [4], it is often advised to limit the use of anonymous classes [4]. Given this recommendation, we found that most of the anonymous classes comprise single methods only (with a profile following Pareto principle [159]). This suggests that developers may not intentionally make code clumsy, rather it is an artifact of the induced application or framework requirements (e.g., SAM types). Besides, developers tend to comply with the advices (e.g., [95]) to avoid deep nesting, thus show a tendency to avoid complexity and achieve a better code structure.

  Moreover, given the advices regarding the use of inheritance, both in favor (due to code reuse facility) and against (due to complexity associated with deeper classes in the inheritance hierarchy [192] and resulting maintenance difficulty [131]), we observed that developers tend to limit the use of inheritance while defining inner classes.

Developers, while using language features, enjoy the flexibility offered to them

We demonstrate that developers employ Java's property mechanism, a feature supported by code convention, as they desire. As a result, a variety of patterns are observed to emerge. We identify a catalog of 10 distinct patterns (described in Chapter 5, Section 5.2.1) that capture different definition of properties developers employ in practice. This catalog shows that the developers utilize, according to their intentions, the flexibility of defining properties offered by Java programming language (unlike built-in support for properties available in other language like C#).

Developers exhibit certain consistency in using language features

We show that there exists a *certain statistical consistency* (as represented by a small and narrow bounded region of computed Gini coefficients) among the developers in employing language features. We show that the distribution profiles of the studied language features are highly concentrated in Java-based software systems, suggesting a consistent practice. We identify a typical comfort range of the studied features within which developers organize solution design. Though any deviation from such observed region does not necessarily imply a problem, the bounded region may be an indicator of some form of cognitive preferences of developers.

But even though the developers concentrate the studied language features in a relatively few number of classes in general, we noticed that the concentration profiles of some of those features (e.g., getter and setter methods) are often negatively related to their proportions profiles. This suggests that there is a tendency in developers to work with small and manageable classes, causing the distribution of features to be dispersed across more classes. This indicates that some sort of God-like aversive design strategy is practiced by developers.

Developers avoid some language features

We found evidence of mismatch between language designers' expectation and developers' application of certain language features. We observed that the concept of local classes are being used very rarely in the software systems of the Qualitas Corpus. Such rare use suggests that this concept is not well-accepted by developers, and also indicates that developers may not use a programming concept unless it offers sufficient value. The rare use of local classes, however, merits an exclusion of this concept from the Java programming language.

Support for language feature modification proposals

We show that, in case of anonymous classes, developers make use of SAM types (abstract classes or interfaces that comprise only one abstract method) substantially with a profile that follows Pareto principle [159]. This indicates that if similar functionality could be implemented with more concise yet effective language constructs, developer burden could be substantially scaled down. As SAM types are one central aspect of lambda expressions [20], the proposals (e.g., [182]) suggesting anonymous classes to be replaced with lambda expression will benefit developers to write more concise and readable program code.

Classification of the software systems in Qualitas Corpus

We classified the software systems in the Qualitas Corpus based on their respective domains. In particular, we identified 12 major domains (e.g., middleware, database). This classification, presented in Chapter 3, Section 3.2.2, can help understanding diversity of the software systems in the Qualitas Corpus, and can also assist studies that involve domain-specific characterization of developer behaviors in using programming language features in Java-based software systems.

Support for study of programming language features

We contribute to the research community an extensible framework for metrics extraction and processing: *jCT - a Java Code Tomograph* [139]. It can be used to assist studies of developer behaviors in using programming language features, and also studies that involve software metrics data. Though jCT is built for Java, similar approach can be adopted to study other languages (e.g., C#). In addition, jCT offers inequality-based measures (e.g., the Gini coefficients, Lorenz curve), as recommended by Vasa [216], to support software metrics analysis.

1.4 Structure of the Thesis

This thesis is organized into a set of chapters and an appendix. In this section, we present a brief description of rest of the chapters. The key content of each chapter is depicted in Figure 1.4.

In **Chapter - 2 : Background and Motivation**, we discuss the scope of this thesis and cover the state-of-the-art to motivate this study.

In **Chapter - 3 : Methodology**, we discuss our research settings. In particular, we describe the experimental data set and also the extrac-

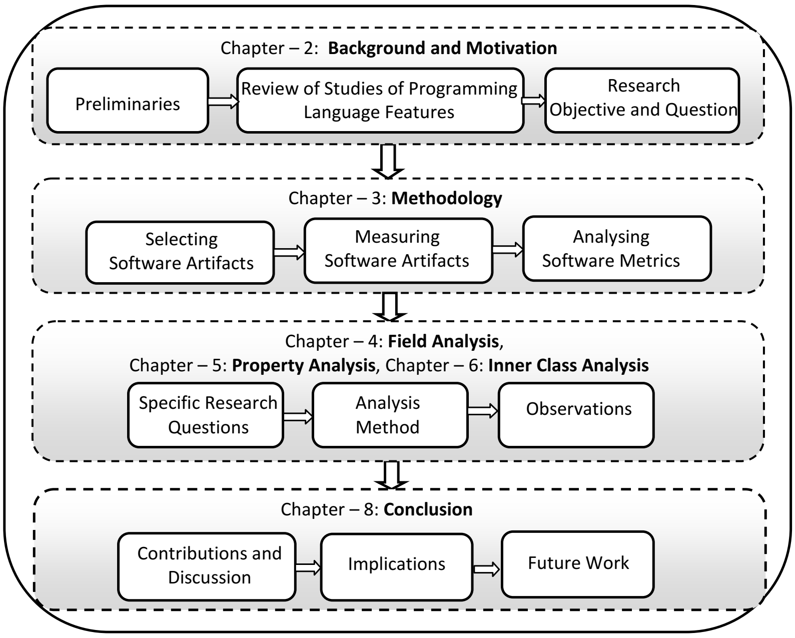

Chapter – 2: **Background and Motivation**
Preliminaries
Review of Studies of Programming Language Features
Research Objective and Question
Chapter – 3: **Methodology**
Selecting Software Artifacts
Measuring Software Artifacts
Analysing Software Metrics
Chapter – 4: **Field Analysis**,
Chapter – 5: **Property Analysis**, Chapter – 6: **Inner Class Analysis**
Specific Research Questions
Analysis Method
Observations
Chapter – 8: **Conclusion**
Contributions and Discussion
Implications
Future Work

Figure 1.4: Thesis Structure - Overview

tion, processing, and aggregation techniques of software metrics data necessary to achieve our research objectives.

In **Chapters - 4 (Field Analysis), 5 (Property Analysis), and 6 (Inner Class Analysis)**, we present the results of our investigation as observations that entail developers behaviors in employing fields, properties, and inner classes, respectively, of the Java programming language.

In **Chapter - 7 : Conclusions**, we present the summary of this thesis, discuss the contributions, implications, limitations, and identify the possible scope of future work.

In **Appendix**, we present *jCT- A Java Code Tomograph* - an extensible framework that enables us to work with emerging metrics definitions. We developed and used this framework to address metrics requirement in this work.

Chapter 2

Background and Motivation

In this chapter, we present the state of the art in the context of this work. We begin with revisiting some preliminaries in section 2.1 to facilitate setting the foundation of our work. These are the basic process of engineering software systems, underlying domains, programming languages, and coding conventions and design guideline that can affect the use of language features in solution design. We then present available studies on programming language features in section 2.2, where we discuss their four aspects: purpose, methodology used, key findings, and limitations along with the benefits of overcoming them. Based on this discussion, we formulate our research objective in section 2.3. Finally, we conclude this chapter with a summary in section 2.4.

2.1 Preliminaries

Software - An Engineered Product

Software Engineering refers to *"the disciplined application of engineering, scientific, and mathematical principles and methods to the economical production of quality software"* [108]. It is concerned with addressing all the aspects necessary for producing and maintaining software systems that satisfy the requirements as defined for them.

The scope of software engineering covers the complete life cycle of software systems. This includes the methodologies necessary for conceiving, designing, implementing, and maintaining the intended software systems [45, 174]. Each of these constituting areas integrates a variety of engineering methods and has to adhere to certain conventions and techniques in order to accomplish the desired tasks of building software systems.

The purpose of software engineering is to navigate within a *solution space* to seek out the most efficient and cost effective solution for a given set of requirements. Given a particular problem, the search in a solution space begins with the acquisition, analysis, modeling, and verification of the requirements of the associated stakeholders (e.g., clients, users, etc.) [102, 160]. In this process, a variety of approaches (e.g., interviews, questionnaires, organizational document analysis) are being employed [160, 235] in order to define appropriate system boundaries.

The gathered requirements are being used to devise specifications comprising different aspects (e.g., data, functional, behavioral) of a desired system. This assists in mapping the requirements to a design space in order to structure and organize a model of the intended solution design. This model captures appropriate data structures and algorithms that satisfy the constraints imposed by the underlying operating environment. The physical representation of the model is the end product - a software system.

A software system is defined as *"computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system"* [111]. Based on the purpose it serves, a software system is focused on capturing a solution design of a particular problem domain.

Domain

A *domain* is a *problem space* that enjoys a set of common features, properties, activities, terminologies, and functionalities. Estublier et al. [75] defined domain as *"an area in which a number of stakeholders*

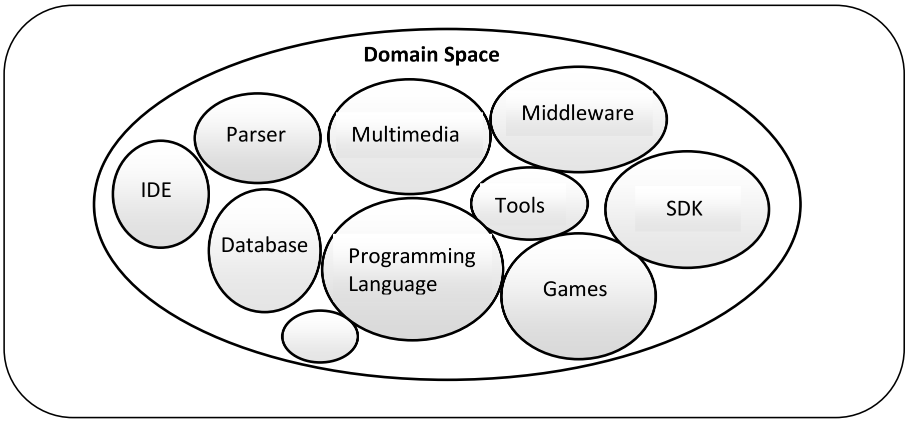


Figure 2.1: Possible solution domains

is repeatedly performing similar activities". A stakeholder can be a person, a team, or a company that are all interested in conducting common activities. In a domain, the stakeholders are involved in activities that share common attributes, features, etc. In other words, domain is *"a set of current and future applications which share a set of common capabilities and data"* [118]. It is a family of similar systems having some common characteristics. For example, *database* is a problem domain that involves storing information to and retrieving it from computer programs. Therefore, each database application (e.g., derby[1], hsqldb[2]) has a similar nature and shares common attributes. Figure 2.1 depicts a set of different domains in the context of software engineering.

In the object-oriented context, a problem space is mapped to the corresponding *object space* through *domain analysis* [60]. Domain analysis is a process of identifying the commonalities and variabilities of the particular domain. It captures *"the activity of identifying the objects and operations of a class of similar systems in a particular problem domain"* [158]. A more appropriate definition related to the field of software engineering is provided by Arango [25]: *"domain analysis is the identification, acquisition and evolution of reusable information on a problem domain to be reused in software specification and construction"*. That is, gathering the common information from a problem area in order to construct software systems.

[1] http://db.apache.org/derby/
[2] http://www.hsqldb.org/

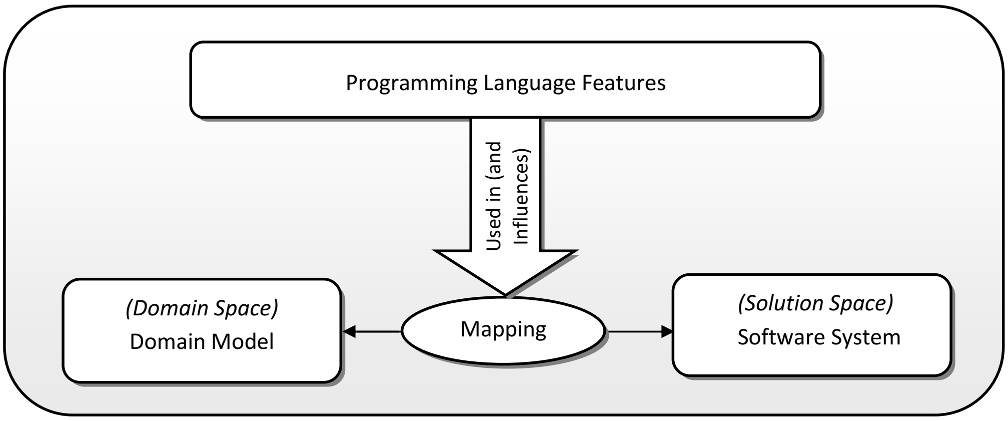


Figure 2.2: Conceptual model : Mapping domain space to solution space

The outcome of domain analysis is a *domain model*. A domain model is *a problem-oriented architecture for the application domain that reflects the similarities and variations of the members of the domain*. A domain model describes all the entities, their properties, and relationships among the properties. A domain model is mapped to concrete software systems by using programming language features that capture domain abstractions (cf. Figure 2.2).

Programming Language

Programming languages[3] are the means for expressing computations. To articulate desired computations, programming languages usually offer different concepts. These programming concepts are organized around a *computation model* (also called *thought model* [180]) that provides us with the *abstract means* to capture computations.

There are many different computational models available, each with its own distinct *form* for representing desired computations and each comes with its own programming techniques. For example, while object-oriented techniques provide us with the means to express desired com-

[3] A programming language is defined as *"a language used to express computer programs"* [111]. Though this definition covers a wide family of languages (e.g., low-level, high-level), we use the term "programming language" to refer to high-level languages (e.g., C, C++) in this work.

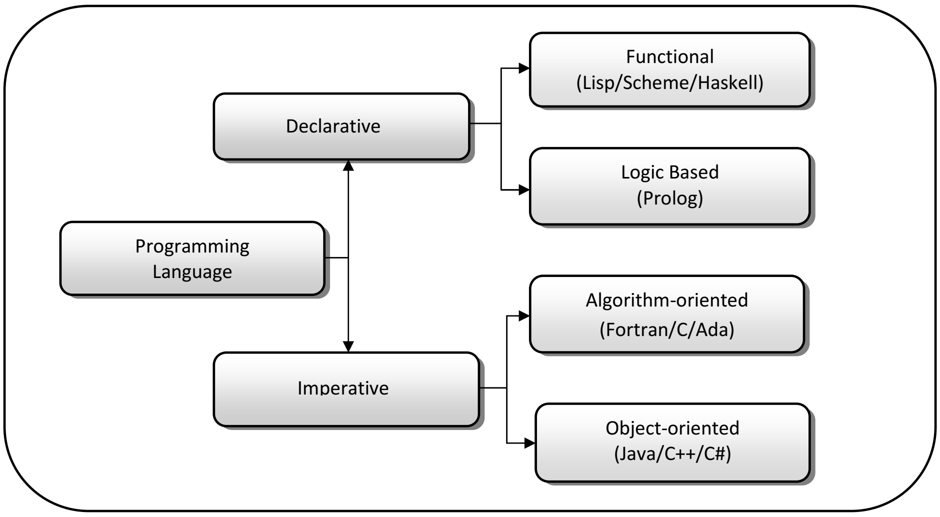


Figure 2.3: Programming language classification [180]

putations based on the use of state and inheritance, functional programming encourages the use of higher order abstractions (e.g., monads and currying), and logic based languages (Prolog) support the use of *Horn* clauses [215].

Based on the thought models, programming languages can be classified into two general categories: imperative and declarative (cf. Figure 2.3) [180]. Imperative languages cater for computations as transformation of states as a process in time, whereas declarative languages usually offer a set of stateless mathematical functions for expressing desired computations.

However, a programming language does not only evolve around a computation model. It is also designed to address the issues of an intended domain. The *domain knowledge* is embedded into the software systems through a programming language. A particular programming language can capture only a specific domain abstraction or it can be independent of any specific domain. Based on the focus of captured abstractions, programming languages can be either general purpose (GPL), or domain specific (DSL) [147].

A general purpose programming language provides powerful abstraction mechanisms, independent of any particular domain. In general, the available syntax and semantics of a GPL are domain agnostic. This enlarges the scope of a GPL, and thus enables us to devise solutions of problems that originate from a wide variety of domains. Examples of widely accepted general purpose programming languages include C++, Java, and C#.

General purpose programming languages, though similar in nature, are equipped with different and often disjoint sets of language features to serve a range of domains. For example, C can be used for system programming purposes (e.g., writing compiler, operating systems) as well as for building a game engine or an accounting software system.

A domain specific programming language, on the other hand, caters to problems of a particular domain. Deursen et al. [212] defined a *domain specific* language as follows, *"A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain"*. That is, a DSL is more specialized than GPL and thus provides high-level abstractions through domain specific constructs and notations. Examples of commonly available DSLs include SQL (database queries), HTML (hypertext markup), and VHDL (hardware design), MATLAB (technical computing), and LaTeX (typesetting).

Still, regardless of the type of programming language, many different additional guidelines [10, 183] are often provided to facilitate the use of available constructs. Such guidelines include *coding standard and conventions* for a particular programming language and *experts advices* on how one should employ the features available in that language during software development. We discuss such conventions and guidelines in the following section.

Coding Standards and Design Guidelines

In this work, we consider coding standards as a broad umbrella term that covers conventions and expert advices in relation to language features in order to encourage better programming practices.

A coding standard, as defined by Howles [107], *defines the format and gives a consistent structure to the code.* The purpose is to guide developers to a common style. While some standards [151, 152] have been developed with a certain context (e.g., safe use of C/C++ in critical systems) in mind, some are optionally provided with programming language specifications. For example, the C# language specification does not define any coding standard [15] but the Java programming language provides us with specific coding conventions to follow [10].

Coding conventions can often be used as alternative means to accomplish the flavor of built-in language features. Consider, for example, the mechanism to allow access to the values of private fields of an object. To achieve this, C# provides built-in support to read, write, or compute associated fields through properties. Java, on the other hand, relies on specific naming conventions to allow the developers to define the required properties through a pair of public methods, namely getter and setter methods [10].

Moreover, coding conventions can also be used as an alternative to configuration files (i.e., *convention over configuration*) to implement a particular functionality. When the associated conventions are being adhered to, the underlying system can extract required information from them, and thus developers' burden to deal with additional configuration files is reduced. For example, instead of employing an XML-based configuration file, conventions can be used for defining beans (and associated autowiring purposes) in the spring framework. Many other frameworks (e.g., Ruby on Rails) encourage the use of conventions, and thus tend to comply with principles like *write beautiful code favoring convention over configuration* [22].

A failure to comply with designated coding standards (when available) can incur inconsistency in program code. For example, consider a variable name that represents the number of lines in program code. The name of such variable can appear in different forms, for example, *line_count, linecount, LineCount*, as a result of an associated developer's style. Though all of those names are valid from the perspective of the language, different naming conventions can result in maintenance overhead, particularly in large projects where hundreds of developers work together. Moreover, software systems are rarely maintained for their whole lifetime by the same original author [10]. Therefore, it often becomes harder for new developers to comprehend program code that does not comply with associated coding standards.

Adherence to coding standards, on the other hand, can yield a number of benefits. These include (i) code consistency (as everyone follows the same standard), (ii) lower learning curve of program code (arising from better readability and comprehensibility), (iii) reduction of the likelihood of potential bugs (as a result of clarity in code). These features, in turn, improve code quality, resulting in increased maintainability. As almost 80% of the lifetime cost of software systems are associated with maintenance [10], adherence to coding standards is strongly encouraged to reduce such cost [16, 78].

However, in addition to the coding standards and conventions, there are expert advices available in the literature (e.g., [85, 105, 163, 183]) that are intended to support good programming practices and also to assist better (in terms of maintainability, for example) solution designs. For example, Gamma et al. [85] suggest to "favor object composition over class inheritance". Parnas [163] emphasizes to keep object representation hidden. Such advices, however, can regulate the use of some language constructs. For example, hiding object representation may cause an increased use of private visibility modifiers of fields, and less frequent appreciation of the non-private ones.

Summary

Software systems are engineering products, and are build to address the necessities of different domains (e.g., database, middleware). The key to map a problem domain to working software systems are programming languages that are equipped with suitable features. The available features provides the developer with the necessary flexibility to express their design decisions. To facilitate the use of such features, different coding standards and expert advices are often provided. However, do developers actually use and adhere to coding standards and conventions? Are there specific preferences in how they employ language features driven by the choice of coding standards?

2.2 Studies of Programming Language Features

The software engineering literature covers a diverse set of issues in the context of studying programming language features and associated design decisions of the developers. To facilitate our understanding of current practices in this context, we discuss the related work from the following four specific perspectives:

- Purpose - What is the focus of a particular study?
- Methodology - What research approach and data analysis techniques are undertaken?
- Findings - What are the key observations?
- Knowledge gaps - What has not yet received much attention?

Purpose

The available work (e.g., [37, 41, 49, 50, 50, 62, 65, 93, 98, 103, 125, 141, 143, 145, 171, 206–209, 224, 225]) on programming language features has different objectives in general. However, based on the primary focus, we categorize it into the following two key areas:

- Use of language features in software development, and
- Impact of language features on software quality attributes (e.g., maintenance, reliability).

The trend of studying how programming language features (e.g., inheritance, encapsulation) are being used by developers has been an active area of research. A common theme of such research is to mine the products built by the associated developers to reveal their design decisions. In this context, many researchers (cf. Table 2.3) empirically investigated the usage of features available in various programming languages. For example, Tempero et al. [77, 208, 209] studied how different features (i.e., fields, inheritance) available in Java are being used by developers, Callaú et al. [49] examined the usage of dynamic features provided by Smalltalk, and Counsell and Newson [62] investigated the use of *friends functions* in C++.

In addition, programming language features are also studied in order to assess their quality impact, where the key interest is to discover any potential relation between the use of particular language feature and quality attributes (e.g., maintenance, reliability). For example, Cartwright and Shepperd [50] investigated the quality impact of inheritance in C++-based software systems, and found substantial differences in defect densities between classes that employ inheritance and and classes that do not.

Methodology

Typical methodologies adopted in the above studies include

- A survey that involves the collection of necessary data through communicating with the developers in form of questionnaires, interviews, etc. For example, Gorschek et al. [93] used on-line questionnaires in their study to collect necessary data.
- An empirical investigation that involves mining necessary software metrics data from a collection of software systems. The size of such

Table 2.1: Data set used in Different Studies

Study	System(s) Investigated
Cartwright and Shepperd [50]	1
Wang and Hou [225]	1
Mancl and Havanas [143]	1
Subramanyam et al. [202]	1
Harrison et al. [98]	2
Harrison et al. [98]	2
Counsell and Newson [62]	4
English et al. [73]	4
Bhattacharya and Neamtiu [37]	4
English et al. [72]	13
English and McCreanor [74]	20
Holkner and Harland [103]	24
Melton and Tempero [145]	81
Tempero et al. [209]	93
Tempero [207]	100
Tempero et al. [208]	100

collection may vary depending on the type of study (i.e., while a large scale study can examine hundreds of software systems, a case study can focus on only one of them). Table 2.1 presents an overview of different studies and corresponding number of software systems investigated.

To analyze the mined software metrics data, different statistical techniques (cf. Table 2.2) are often used. We classify the primary analysis approaches into the following two categories:

- Frequency Distribution Analysis:
  This involves an analysis of a particular attribute of interest by counting its frequency of occurrence in a given entity. For example, while Tempero [207] investigated field (attribute) usage pattern in Java-based software systems (entity), Callaú et al. [49] investigated the use of dynamic features in Smalltalk-based systems using frequency count (e.g., percentage of the use of a particular attribute).

- Evaluation of Hypothesis:
  This entails a formulation of specific hypotheses and their evaluation using different statistical techniques (cf. Table 2.2). For example, while English et al. [72, 73] used the *Pearson Chi-square Test* for hypothesis testing, Bhattacharya and Neamtiu [37] performed *Linear Regression Analysis* for that purpose.

Table 2.2: Statistical Techniques Used in Different Studies

Statistical Techniques	Studies
Frequency Analysis	Tempero et al. [207–209], Callaú et al. [49], Wang and Hou [225]
Spearman Rank Correlation	English et al. [72], Cartwright and Shepperd [50]
Pearson Correlation	Counsell and Newson [62]
Linear Regression	Cartwright and Shepperd [50], Bhattacharya and Neamtiu [37]
Chi-square Test	Melton and Tempero [145], English et al. [72], Harrison et al. [98]
t-test	Bhattacharya and Neamtiu [37]

Key Findings of Available Work

The available studies provide us with rich body of knowledge regarding the usage and quality impact of programming language features. We summarize their key elements in Table 2.3.

Table 2.3: Studies on Programming Language Features

Study	Language	Investigation	Findings include:
Tempero et al. [209]	Java	Usage	Most types (classes and interfaces) in Java-based software systems are relatively shallow in the inheritance hierarchy.
Tempero [207]	Java	Usage	Though the developers define non-private fields, they are less likely to use them.
Tempero et al. [208]	Java	Usage	Found evidence of substantial overriding practices (i.e., most of the subclasses in a software system override at least one inherited method).
Callaú et al. [49]	Smalltalk	Usage	Dynamic features are rarely used in practice
Briot and Guerraoui [41]	Smalltalk	Usage	Rich and reusable libraries of classes, together with the flexibility offered by Smalltalk, allow the language to be a very good foundation for concurrent and distributed programming.
Counsell and Newson [62]	C++	Usage	Friend functions are extensively used for facilitating global operator overloading functions.
Cartwright and Shepperd [50]	C++	Usage	Limited use of object-oriented language features such as inheritance and polymorphism in the investigated C++ software system.
Wang and Hou [225]	C++	Usage	The most advanced nature of function overloading tends to be defined in only a few utility modules (in the systems programming area of the investigated software systems, Mozilla).

English et al. [72]	C++	Usage	The use of the friend constructs is independent of other class design issues. Moreover, this study also confirm that there is no link between friend constructs and inheritance, contrary to findings (i.e., friend constructs might be used as an alternative to inheritance) by Counsell and Newson [62].
Voigt et al. [224]	Java and C#	Usage	Inconsistent programming practices are found regarding the use of encapsulation concept. Moreover, object encapsulation is found to be more intuitive and also provides object-oriented design advantages when compared to class encapsulation.
Tempero et al. [206]	Java	Usage	There is a significant amount of unused code in the investigated software systems.
Melton and Tempero [145]	Java	Usage	Classes with a non-private static method or field (which is accessed from another class) are likely to be involved in dependency cycles.
Knuth [125]	Fortran	Usage	Typical use frequencies of Fortran constructs for supporting compiler design/optimization.
Malayeri and Aldrich [141]	Java	Usage	Java programs could be somewhat improved by using structural subtyping.
Gorschek et al. [93]	OO Concepts	Usage	The investigation focused on understanding how developers make design decisions (i.e., how developers understand and apply available theory and advices on good object-oriented design. While developers follow the advice on hiding representation, they tend to violate advices regarding class size and depth.
Muschevici et al. [155]	CLOS, Dylan, Cecil, Diesel, Nice and MultiJava	Usage	There are potential demand for multiple dispatch in Java programs if it is supported by the language.
Holkner and Harland [103]	Python	Usage	Dynamic features, particularly non-reflective dynamic features (e.g., using objects dynamically and dynamic code execution) are being used in all studied programs. About 70% of these programs have less dynamic activity after startup. This indicates that *RPython* - a subset of Python that is statically typed [24], can be used as replacement for Python in some cases. RPython permits a full set of Python features to be used up to a certain point (e.g., class initialization), then a restricted set of features may be used for achieving runtime efficiency.
Cartwright and Shepperd [50]	C++	Quality Impact	There exists a substantial difference in defect densities between classes with and without the use of inheritance. Classes employing inheritance are three times more defect-prone than classes without inheritance structure.

Daly et al. [65]	C++	Quality Impact (Maintenance)	Software systems with a hierarchy of 3 levels of inheritance depth are approximately 20% quicker to maintain than software system without inheritance. This observation, however, is based on an investigation of only 2 small programs (less than 450 lines of code) written in C++. Besides, the maintainability is checked by 31 students. Therefore, the conclusion may not represent the true impact of inheritance on maintenance of contemporary software systems.
Harrison et al. [98]	C++	Quality Impact (Maintenance)	Software systems that do not employ inheritance are comparatively easier to modify than software systems comprising more involved inheritance hierarchies.
Bhattacharya and Neamtiu [37]	C/C++	Quality Impact (Maintenance)	Programs written in C++ are less prone to bugs and also have a higher internal quality when compared to programs written in C.
Mancl and Havanas [143]	C++	Quality Impact (Maintenance)	The use of object-oriented features resulted in maintenance benefits.
Ponder and Bush [171]	Smalltalk	Quality Impact/Usage	Polymorphism can significantly affect quality attributes (i.e., program understandability) if abused.
English et al. [73]	C++	Quality Impact/Usage	Classes declared as friends have higher coupling than classes not declared as friends. Furthermore, the number of friends of a class influence the number of protected and private members in it
English and McCreanor [74]	Java and C++	Quality Impact	C++ systems might be more difficult to maintain or comprehend than Java systems.
Holtz and Rasdorf [104]	Fortran 77, Pascal, C, Modula-2	Quality Impact	Programming constructs available in different languages can contribute to code complexity and understandability.
Subramanyam et al. [202]	Java and C++	Quality Impact	Java classes that exhibit higher values of DIT metric and C++ classes that comprise higher values of CBO metric are associated with higher number of defects

The observations presented in Table 2.3 suggest a variety of issues regarding language features. They are twofold: issues discussed in the context of language features supported by a language, and issues related to incorporating new features to a language.

An available language feature can give rise to problems. For example, rare use of dynamic features (e.g., behavioral and structural reflections) in Smalltalk [49]) indicates a potential mismatch between the language designers' expectation regarding adoption of such features by developers and the actual acceptance by them in practice. Similarly, the advanced nature of function overloading [225] in system programming area suggests its usefulness in specific domains. A list of similar indications is presented in Table 2.4.

Table 2.4: Indications associated with use-case of language features

Language Feature(s)	**Indications based on Use-Case**
Dynamic Feature [49]	Mismatch between language designers expectation and developers practices
Function Overloading [225], Friend Functions [62], Rich language abstractions [41]	Usefulness of certain features in specific domain (or for specific purpose)
Method Overriding [208]	Well-accepted programming concept (as developers use them extensively)
Inheritance [209]	Use of programming concepts in a certain style (e.g., preferred inheritance hierarchy is shallow)
Encapsulation [224]	Inconsistent programming practice

Moreover, complex language abstractions may cause developers to incorrectly use them. For example, higher depth of inheritance (DIT) is associated with higher defects [50, 202]. This suggests that the use of inheritance to construct deep class hierarchies may be somewhat difficult, causing possible chances to introduce faults when using inheritance at deeper levels. The findings also indicate that the available language abstractions may ease (e.g., [143]) or complicate (e.g., [74, 98, 104, 171]) different types of work (e.g., writing, understanding and maintaining code) for developers.

Adding new features to language may improve productivity. For example, Malayeri and Aldrich [141] investigated the usefulness of *structural subtyping* in Java. Structural subtyping is a form of typing where the subtype relationship is determined based on the definition or structure of the type. More precisely, X is a structural-subtype of Y when every abstract member of Y has a matching member in X [142]. This feature is available in languages like *Objective Caml* [130] and *PolyToil* [44]. The study concluded that Java programs could be somewhat improved by using structural subtyping in some cases. They found, for example, 32% usage of reflections (involving `Class.getMethod`) can be rewritten using structural downcasts.[4] They summarized that developers often emulate structural types through the use of reflection.

[4] Downcast means casting a reference of a base class to one of its derived classes.

Another example is the *multiple dispatch* - a feature that considers more than one argument of a method while determining the target method for execution. This feature is available in languages like *Cecil* [51], *Diesel* [52], *MultiJava* [59]. But commonly known object-oriented languages (e.g., Java) offer single dispatch mechanism where a target method is determined by dynamic type of only its first argument (i.e., method receiver, *this*). Muschevici et al. [155] investigated Java programs to determine scope to use multiple dispatch. They found that this feature is often being simulated by using cascaded `instanceof` - a construct used for runtime type testing. They suggested that multiple dispatch is beneficial as it can assist to improve the expressiveness of a language by providing first class alternative to cascaded `instanceof`, and concluded that Java programs could benefit from multiple dispatch if it were provided.

Knowledge Gaps

The outcomes of the studies summarized in Table 2.3 do not include *developers preferences in using language features that are associated with available coding standards and conventions (e.g., properties in Java).* Another key limitation of the existing studies is that they provide us with insufficient information regarding *a systematic approach to study the use of programming language features in the context of coding conventions.*

We can only succeed to assist developers with better language constructs for building software systems if we have comprehensive knowledge on all aspects of the concepts available in programming languages. These include not only the knowledge on language features and their applications in practice, but also the developers preferences in adhering to available coding standards, experts advices and design guidelines. Therefore, it would be useful to study language features from this perspective with an aim to bridge the gaps that remain unanswered.

The outcomes of such study could be useful from a number of different perspectives. One of them is programming language design. As language design is an experimental task (where each prototype implementation is often improved to reflect developer preferences and evolving necessities), the insights gained from this study may be used to support programming language design and evolution. In particular, the outcomes may support the notion of evidence-based language design [155] - retaining or refining language features based on evidences of use-cases.

Moreover, the outcome could assist software managers, maintenance, and quality assurance personnel with insightful information on developer tendencies in adhering to coding standards and design guidelines. As consistency in developer practices assists organizations in improving software quality [30], they can adopt regulatory actions based on the resulting outcomes of this study.[5]

2.3 Research Objective

Developers are influenced directly or indirectly by available coding standards, conventions, and expert advices. Although prior work in this space informs us about the use of language features, their quality aspects, and a little about tendency in following expert advices, we do not have a comprehensive understanding on how developers are directed by the recommendations offered in the conventions. Our work focuses on this gap. In particular, we address the following question:

> *What is the nature of developers behaviors in using programming language features in response to the influence of available coding standards, conventions, and design guidelines?*

The investigation of this question involves many different relevant entities. Figure 2.4 provides us with an overview of such entities, and identifies which specific ones are involved in this study. This figure depicts a typical scenario of the interactions among the entities, and thus covers a wide variety of languages and coding standards that one can

[5] A more detail discussion is presented in conclusion chapter (implication section).

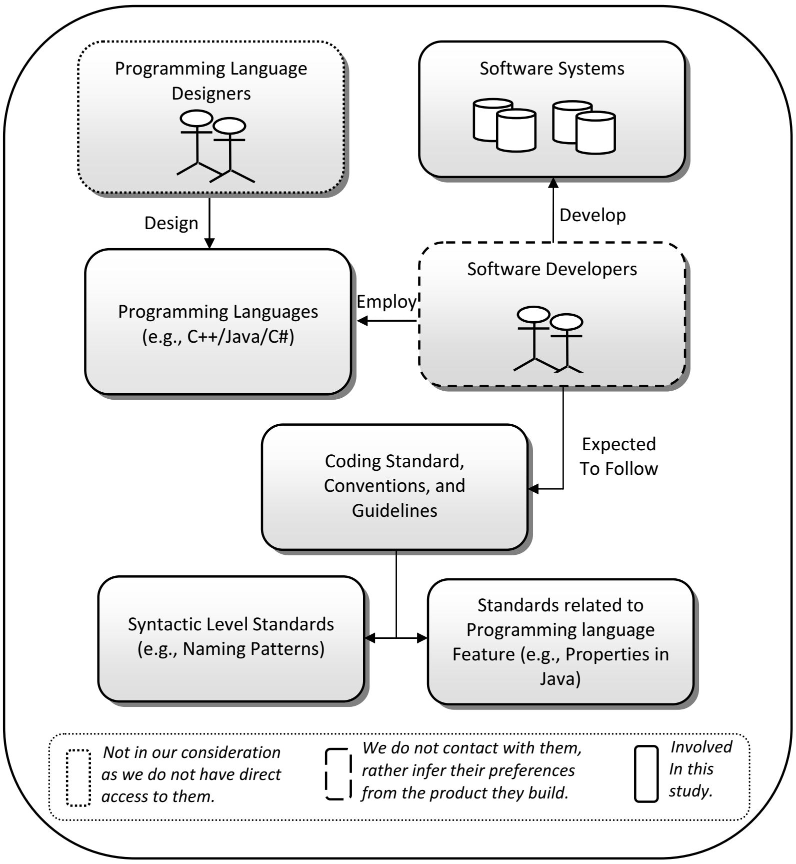

Programming Language Designers
Software Systems
Design
Develop
Programming Languages (e.g., C++/Java/C#)
Employ
Software Developers
Expected To Follow
Coding Standard, Conventions, and Guidelines
Syntactic Level Standards (e.g., Naming Patterns)
Standards related to Programming language Feature (e.g., Properties in Java)
Not in our consideration as we do not have direct access to them.
We do not contact with them, rather infer their preferences from the product they build.
Involved In this study.

Figure 2.4: Entities involved in our study

consider to answer the question asked above. But rather than looking at different languages, we aim at answering this question in the context of one specific language, Java, and generalizing our observations for similar languages. This decision allows us to conduct a more focused and extensive investigation.

Java is one of the most widely used programming language in the object-oriented programming realm. It has been used to develop both commercial and open source software systems.[6] Moreover, an increasing number of researchers (e.g., [35, 144, 145, 156, 208, 209]) are found to be interested in studying this language, indicating its acceptance in the research community. The wide acceptance of this language in both academia and industry motivated us to study this language.

[6] See http://www.sourceforge.net and also [77] for collection of such software systems.

The Java programming language is equipped with a wide variety of features. However, rather than investigating them all in one study, we decided to explore any small subset in depth. Moreover, not all of the features available in Java are related to the question we aim to answer. Therefore, it is necessary to decide on which features to focus when describing the influence of coding standards and conventions on developer preferences. This requires selecting a certain set of features that are relevant, unexplored, and associated with coding standards, conventions and guidelines, respectively.

Given this context, we considered two aspects of the object-oriented programming paradigm (cf. Figure 2.5). These are (i) object state management, and (ii) object behavior implementation. The Java programming language offers abstractions for serving both of these purposes.

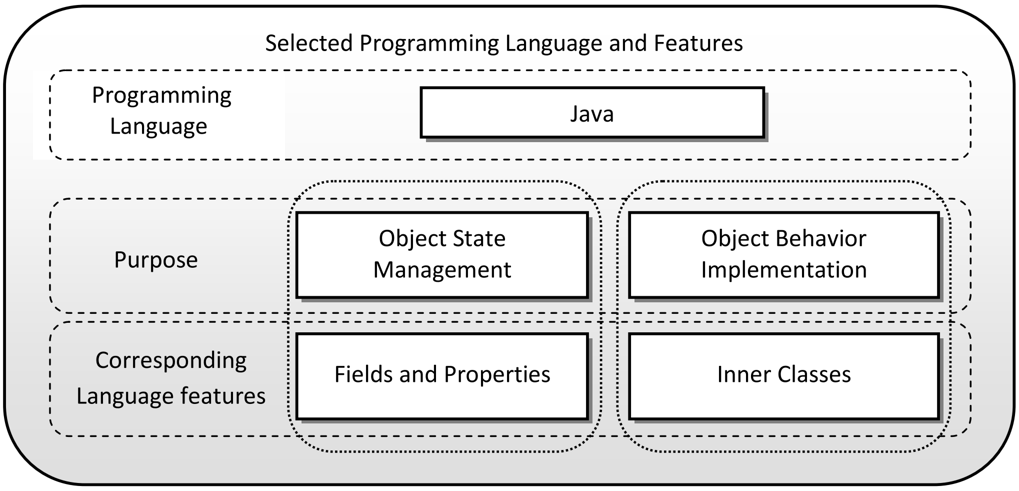

Selected Programming Language and Features
Programming Language
Java
Purpose
Object State Management
Object Behavior Implementation
Corresponding Language features
Fields and Properties
Inner Classes

Figure 2.5: Selected language features

In Java, the state of an object is represented by fields. Access to fields is regulated by visibility modifiers (e.g., public, private). To shield internal state of objects from external access, associated fields are often made private. A private field can only be accessed externally or modified through the property mechanism (i.e., so-called getter and setter methods). As Java lacks built-in support for the property mechanism, developers have to adhere to coding conventions, which requires a property to be signified by two components [71, 153]: the field name associated with the property and the prefix “get" and “set" to denote a getter and setter, respectively. If the name of a field is *Color*, then the name of the corresponding getter and setter methods are *getColor* and *setColor*.

While fields and properties are involved in managing state of objects, there are language constructs for implementing the behavior of objects. One such construct is the notion of inner classes that are predominantly used to structure events in domain models. In particular, it is generally accepted that inner classes are designated primarily for developing adapter classes [4], though other types of usage are possible. They also offer convenient mechanism (i.e., callback facility - programmatically privileged relationship between an inner class and its enclosing class) for developing event-based systems.

The underlying reasons for selecting properties and inner classes include the following:

- Limited empirical evidence exist on the use of these features, particularly properties and inner classes, in Java-based software development.
- Little is known about the how developers utilize coding conventions in accomplishing the flavor of built-in language feature (i.e., while properties in Java are convention, they are built-in language feature in C#).
- There is debate surrounding the use of inner classes, particularly anonymous classes, in Java-based software development (i.e., whether anonymous classes should be replaced with lambda expressions [182, 184]).

The use of the selected features can be affected by different guidelines (e.g., [4]), opinions (e.g., [105, 183]), and principles (e.g., *Tell, Don't Ask* [109, 110], *Law of Demeter* [132]). Table 2.5 presents some insights into these guidelines and thus represents the theoretical expectation regarding the use of selected programming language features.[7] Given

[7] Detail discussion is presented in respective chapters.

such guidelines and opinions, it would be useful to know how developers *actually* comply with them.

Table 2.5: Some advices, which represent theoretical expectation, regarding the use of selected programming language features

Language Features	Advices/Guidelines/Conventions
Fields	(i) *All data should be hidden within its class [183]*, (ii) *Don't expose state if you don't have to* [12].
Properties	(i) *Do not change the state of an object without going through its public interface [183]* - suggests to use getter and setter methods. (ii) *Getter Setters are evil* [105], (iii) *Tell, Don't Ask* principle [109, 110] - avoid using getter methods, (iv) *Law of Demeter* [132] - avoid getter methods.
Inner Classes	(i) *Limit the use of anonymous classes* [4], (ii) *Anonymous classes can make code difficult to read* [4], (iii) Anonymous classes are considered more verbose and extremely clumsy as they have bulky syntax [184], and therefore can make code somewhat difficult to read.

As adherence to the available coding standards and advices results in many benefits (e.g., better readability, comprehensibility, and maintainability of program code [78]), we can expect that developers, in general, follow the standards offered to them. But we have little empirical evidence to test such expectations against developer tendencies in practice. In this work, we attempt to understand developer behaviors in this context. In addition, we demonstrate different aspects of their behavior that are associated with the use of selected language features.

In particular, we make our initial research question more concrete and ask

> *What is the nature of developer behaviors in using fields, properties, and inner classes of the Java programming language with respect to the influence of associated coding standards, conventions, and design guidelines?*

2.4 Summary

The available programming languages and their associated features have been studied by many researchers (e.g., [50, 50, 77, 145, 202, 206, 208]). Their investigations are mainly concerned with two different facets of *available* language features: *usage patterns* and *quality aspects*. There is another arc of similar studies [141, 155] that investigated the scope to incorporate *new features* by identifying use-cases of certain constructs that appear to be an emulation of features available in other languages.

However, the key limitations of available studies is they do not provide us with enough insights into developer tendencies in practicing language features under the influence of available coding conventions and guidelines. Moreover, we do not have a systematic method to study language feature in this context. These additional details are necessary in order to raise our capability to study language features, and also to enrich our understanding on developers preferences in employing language features in solution design from the perspective of coding conventions and guidelines.

Chapter 3

Methodology

The nature of our work motivated us to select an empirical research approach that involves collecting information from experimental data set, and summarizing the distilled data with suitable aggregation techniques. This approach is presented in this chapter by decomposing it into the following two facets: (i) selecting software artifacts, and (ii) measuring software artifacts. In the first facet (section 3.2), we present the context of our experimental data set and selected software artifacts. In the second facet (section 3.3), we describe how we measured the selected software artifacts. This involves the process of defining, collecting, and analyzing necessary software metrics. We conclude this chapter by presenting a summary of the methodology in section 3.4.

3.1 Introduction

In order to achieve our research objective, we conduct an empirical study. Such study, when compared to an analytical or theoretical one, is considered to be comparatively closer to the real world [98]. Moreover, *"empirical studies play a fundamental role in modern science, helping us understand how and why things work, and allowing us to use this understanding to materially alter our world"* [166].

An empirical study can be conducted by adopting either *qualitative* or *quantitative* research methods, or even both, depending on the problem specific settings [231]. To yield a valid conclusion, a qualitative method relies on materials in the form of results of interviews, surveys, questionnaires, observations derived from non-numerical and textual data, images, and so on. [149, 188]. On the other hand, a quantitative method completely depends on objective data (often numeric in form) for the same purpose [166].

In this work, we use a quantitative method that involves mining necessary software metrics data from a collection of software artifacts, and analyzing them with statistical methods [166]. The selection of this method is justified by the fact that developer practices are imprinted into the software artifacts that they produce. Therefore, software artifacts can be used as a source of meaningful information regarding what developers *actually* do. Based on this fact, many researchers (e.g., [35, 88, 140, 144, 145, 156, 206–209, 216]) used software artifacts as source of required information in their studies that resulted in valuable insights regarding not only the associated software artifacts, but also the underlying developer design decisions.

Our work involves the following two steps:

- **Selecting Software Artifacts**
  - Identifying a representative collection of software artifacts that can be used as the basis for our observations.
- **Measuring Software Artifacts**
  - Mining the selected software artifacts to extract relevant software metrics that capture the features of programming language we are interested in, and
  - Aggregation and analysis of the distilled software metrics data to achieve our research objective.

3.2 Selecting Software Artifacts

3.2.1 Context

In this study, we investigated open source software systems. In the following section, we provide an introduction to open source software systems and also the underlying reasons for selecting this domain for investigation.

Open Source Software Systems

Software systems can be developed in a different fashion. Raymond [179] describes two styles of software development: the *cathedral* and the *bazaar*. The former refers to the *classical* development model involving a well-organized, full-time development team in a closed environment, and the later is concerned with software development with loosely-organized, volunteer developers to yield open source software systems.

Table 3.1: Criteria of Open Source Software Systems

Criteria	Description
Source Code	The source code of the software system must be available either in any distribution media (with a reasonable reproduction cost in this case) or downloadable from internet without any charge. Obfuscated source code and intermediate forms (e.g., the output of a preprocessor or translator) are not allowed.
Derived Works	Modifications and derived works are allowed by the license.
Integrity of the Author's Source Code	The distribution of modified source code must be allowed. It is required for the derived works to carry a different name or version number from the original software.
Free Redistribution	The software system must be redistributed without any fee.
No Discrimination	No discrimination against any persons, group of persons, or fields of endeavor. That is, it may not limit the usage of the programs in any selected domain.
Distribution of License	The rights attached to the program must apply to everyone to whom the program is redistributed without any additional license.
Technology Neutrality	License must be technology-invariant and must not impose restrictions on other software systems.

Open Source Software refers to software systems that are free, available in source code form, and adhere to the *Open Source Definition* as regulated by the *Open Source Initiative* - a non-profit organization.[1] The definition[2] of open source comprises the criteria described in Table 3.1.

The notion of Open Source Software has attracted an enormous attention in recent years. These software systems are well known today for their adaptability, reliability, and portability [38]. Open source development supports faster system growth, more creativity, more modularity, and is less likely to have defects as these are discovered and fixed rapidly [165]. The development process is empowered by the collaborating developers working on open program code.

The loosely coupled community of developers spread across the world contribute voluntary often without any institutional support [100]. The associated developers are driven by a variety of reasons [100], including social and technological motivations [39]. The developers gain reputation for talented work [39] as the contributed code is visible to members of the community.

Studying Open Source Software

The key motivations for selecting the open source paradigm in our study are (i) permissive licenses that permit study, (ii) access to quality software systems, (iii) a basis to study how developers really tend to employ programming language features to build software systems without any constraints imposed on them, and (iv) an accepted approach by research community (for example, Godfrey and Tu [90] used Linux to study its evolution, Vasa [216] used a collection of 40 open source software systems to study their growth and change dynamics). In addition, the use of open source software systems is also a cornerstone in studies [207–209] that are related to this work. For example, Tempero et al. [209] investigated a corpus of 93 software systems to understand the inheritance structure in Java applications. Such studies indicate that open source software systems can serve as a fruitful basis for a variety of research projects.

[1] `http://www.opensource.org`

[2] `http://www.opensource.org/docs/osd`

3.2.2 Experimental Software Artifacts

The choice of a specific set of software artifacts depends on the research objectives of a study. As our work involves investigation of language features available in Java, we restrict our focus to Java-based software systems only.

A well accepted data set comprising a curated collection of open source Java-based software systems is the Qualitas Corpus [77]. Many researchers have used this data set as reference in their studies (see [13]). However, Qualitas Corpus has different releases. It is therefore necessary, at least to allow for replication of our results [77] to commit to a specific version. Hence, we selected the release 20101126 [177] as primary data set for our study.

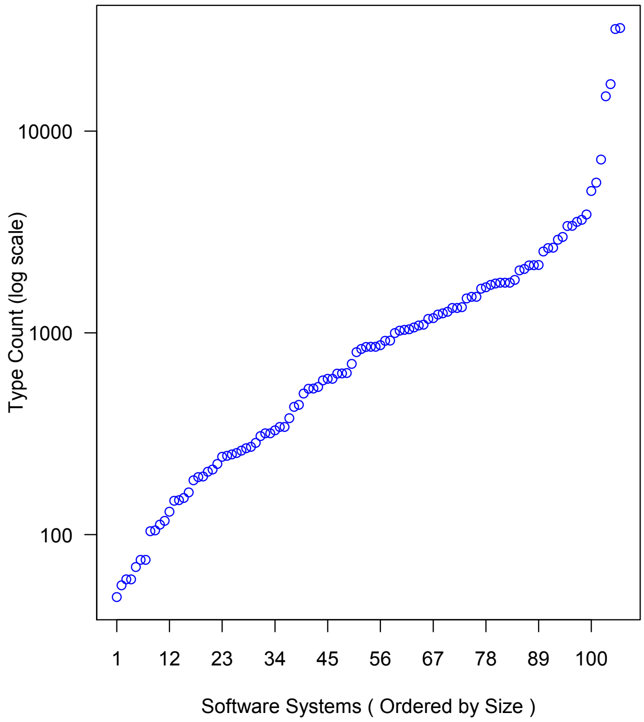


Figure 3.1: System Size Distribution in Qualitas Corpus

The Qualitas Corpus 20101126 provides a curated collection of 106 software systems with varying size. The size is measured as total number of types present in a software system. The size distribution of Qualitas Corpus is depicted in Figure 3.1 and ranges from 49 (jasml) to 32,475 (netbeans) with the median value of 901.

The software systems in Qualitas Corpus are diverse in nature. An overview of the diversity, in terms of domains of constituent software systems, is presented by Tempero et al. [77]. This overview, however, is not detailed enough to allow for studying developer practices in using programming language features in Java-based software systems. In particular, we cannot investigate per se whether the usage patterns of language features vary across domains. For this reason, we developed a high-level domain specific classification of the software systems available in Qualitas Corpus.

Based on the nature of functionality provided by these software systems, we categorize them into 12 different domains. These are (i) Parser/-Generator/Make, (ii) 3D/Graphics/Media, (iii) Games, (iv) IDE, (v) Diagram Generator/Data Visualization, (vi) Database, (vii) SDK, (viii) Middleware, (ix) Server, (x) Programming Language, (xi) Testing, and (xii) Tool. We describe the general criteria used in categorizing the software systems below.

- **Meta-tools (Parser/Generator/Make)**
  A software system that is used to build executable programs from provided source code and associated libraries. In general, such software systems read an associated *configuration file* or *makefile* that comprises the necessary information for producing the corresponding target program. For example, apache ant is used to build Java applications and javacc is used as a parser generator for Java applications. While the former reads build files for constructing executable Java programs, the later processes a grammar specification for transforming it to corresponding Java artifacts.

- **Integrated Development Environment (IDE)**
  A software system that provides comprehensive support for software development activities by integrating the necessary components (e.g., source code editor, corresponding compiler, interpreter, debugger). Moreover, such software systems usually provide generally useful libraries, packages, GUI (graphical user interface) and build automation tools. Popular IDEs for software development include eclipse and netbeans.

- **3D/Graphics/Media**
  A software system that offers functionality like drawing graphics, playing media files, etc. For example, sunflow is a rendering system for photo-realistic image synthesis, joggplayer is a media player, and art of illusion is a 3D-modeller, ray tracer, and renderer.

- **Testing**
  A software system that provides support for software testing activities, either as a stand-alone application or as part of another application. For example, junit is a unit testing framework. It facilitates *test driven development* by supporting automated unit test cases generation. Findbugs is an application that looks for bug through static analysis of source code. While junit can be integrated with IDEs (e.g., eclipse), findbugs works as stand-alone application.

- **Software Development Kit (SDK)**
  A development kit that enables software development activities for a certain target system (e.g., hardware platform, operating system, software framework). It offers, in general, illustrative sample projects, platform or system specific APIs and libraries, documentations, development tools, simulators, and emulators. For example, geotools-2 (gt2) is a library for manipulation of geospatial data for geographical information systems (GIS).

- **Middleware**
  A software system that provides a set of services that allow multiple processes (running on one or more machines) to interact with each other. It comprises a reusable set of abstraction that provides support for application development with a well defined set of libraries and APIs. In addition, it mediates between an application program and a network in order to simplify the development of complex and distributed applications. For example, a software system that facilitates interaction between an application and database servers is a data access middleware (e.g., c-jdbc is a database cluster middleware).

- **Server**
  A software system that provides the functionality of processing requests of, and delivering necessary data to, other client programs (running on one or more machines). For example, jboss is an Java EE-based application server.

- **Diagram Generator/Data Visualization**
  A software system that is focused on presenting data in visual form. This category also includes software systems with the purpose of report generation (e.g., textual, graphical). For example, while jgrapht is a diagram generator, itext is a library that facilitates the tasks of creating and manipulating PDF documents.

- **Tool**
  An application that is focused primarily on accomplishing a particular task or providing specific services (e.g., creating, modifying, analyzing). This includes standalone applications, editors, toolkits, libraries for supporting specific task, etc. For example, while weka is data mining tool, jedit is an editor.

- **Games**
  Software systems which are either games or provide support for developing games. For example, megamek is a turn-based strategy game and marauroa is multiplayer online game engine framework.

- **Database**
  A software system that provide support for data management activities (e.g., data access, update, delete, persistence). For example, apache derby, hsqldb are relational database management systems.

- **Programming Language**
  This category includes different programming languages. For example, aspectJ is an aspect-oriented extension of Java programming language, and JRuby is an Java implementation of the Ruby programming language.

Table 3.2 presents the 12 categories and the list of software systems belonging to them. It should be noted that there are some domains

Table 3.2: Domains of the Software Systems in Qualitas Corpus

Category	**Applications**	**Count**
Parser / Generator/ Make	ant, antlr, javacc, jparse, maven, nekohtml, sablecc, xalan, xerces	9
3D/Graphics/Media	aoi, drawswf, galleon, jhotdraw, joggplayer, sunflow	6
Games	freecol, marauroa, megamek	3
IDE	eclipse, netbeans, checkstyle, drjava, nakedobjects,	5
Diagram Generator / Data Visualization	argouml, displaytag, exoportal, ireport, itext, jasperreports, jext, jgraph, jung, velocity	10
Database	axion, derby, hsqldb, squirrel_sql	4
SDK	colt, gt2, jchempaint, jFin_DateMath, jpf, trove	6
Middleware	c_jdbc, castor, cayenne, hibernate, informa, ivatagroupware, jena, jspwiki, jtopen, myfaces_core, openjms, oscache, picocontainer, tapestry, quartz, quickserver, springframework, struts, xmojo	19
Server	freecs, james, jboss, roller, tomcat, webmail	6
Programming Language	aspectJ, jruby	2
Testing	cobertura, emma, findbugs, fitjava, fitlibraryforfitnesse, htmlunit, jrat, junit, quilt, log4j, pmd	11
Tool	azureus, columba, compiere, ganttproject, heritrix, jag, jasml, jedit, jfreechart, jgraphpad, jgrapht, jgroups, jmeter, jmoney, joggplayer, jrefactory, jsXe, lucene, mvnforum, poi, pooka, proguard, rssowl, sandmark, weka	25
Total		106

(i.e., middleware and tools) that comprise comparatively more software systems. The underlying reasons include both the nature of these software systems, and also the general criteria used in classifying them. We do not make any claim, however, that this classification is perfect. Its primary aim is to serve as a vehicle for studying developer decisions with respect to the use of programming language features. Other interpretations are possible. For example, a further decomposition of the categories into primary and secondary type could accommodate orthogonal system characteristics (e.g., pmd, currently classified as testing tool, has also significant characteristics that warrant a secondary classification: IDE). Such decomposition may give rise to a refinement of our classification but they are not part of this investigation.

3.3 Measuring Software Artifacts

The selected software systems are required to be measured in order to achieve our research objective. In this section, we describe how we measured them. To facilitate our understanding, we divide this section into three different facets (i) the process of measurement - presents fundamentals of software measurement, (ii) software metrics - comprises definition and extraction process of metrics data, (iii) analyzing software metrics - describes techniques used in summarizing metrics, and (iv) inequality measure - a technique used to analyze metrics involved in this work.

3.3.1 The Process of Measurement

A. Core principles

"Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules" [81]. The rules of assignment can include any consistent relations, except random assignment, that capture the underlying intent [117]. Thus, the measurement process entails a mathematical characterization of a particular attribute for different purposes (e.g., better understanding, assessing current state, and improving).

While Fenton & Pfleeger [81] define measurement as *"a mapping from the empirical world to the formal, relational world"*, Kitchenham et al. [122] demonstrate this mapping through a structural model of measurement that captures the objects involved in measurement and the interaction among them (cf. Figure 3.2).

The real world comprises *entity* and *attribute*. While an entity represents an object of real world (e.g., a computer, a software system), an attribute captures a property of that entity. For example, the number of total *line of code* in a software system is an attribute that describes the size of that system. On the other hand, the mathematical world contains a *measurement instrument* that captures the values of different attributes and expresses them in *unit*s of associated *scale type*.[3]

The *scale type* is important in any type of measurement. According to Pfleeger et al. [168] *"unless we are aware of the scale types we use, we are likely to misuse the data we collect"*. For example, central tendency

[3] There are different types of scales available in measurement literature. These are *Nominal, Ordinal, Interval, Ratio* and *Absolute* scale [80, 81, 201].

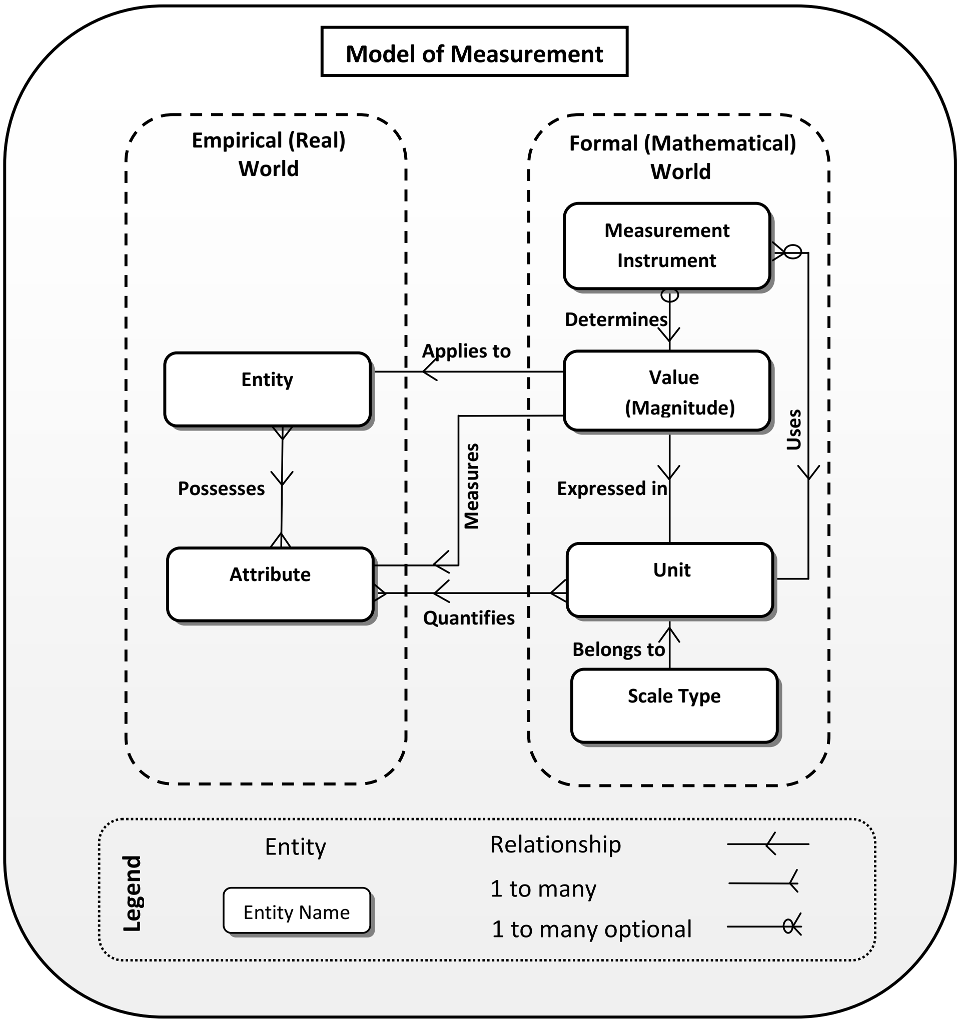


Figure 3.2: A Structural Model of Measurement by Kitchenham et al. [122]

measures (e.g., mean) locate the middle of a data set by dividing the aggregated data by the total number of data elements. But, the notion of mean does not convey any meaningful information if the underlying scale types are *nominal* or *ordinal*. The former assigns labels to data elements whereas the latter uses ranks, and therefore standard descriptive operations (e.g., mean, variance, standard deviation) on these scales are meaningless. While median is appropriate for ordinal scale, it is inapplicable for the nominal one. The ratio scale allows all types of statistical measures [201]. Therefore, one should be aware of scale type and appropriate statistics during software measurement.

In this work, we made use of neither ordinal nor interval-scaled data (as the nature of our study does not demand them). The only use of nominal-scale is the classification of the primary input data set into different domains (e.g., database, games). Apart from this, our metrics data processing involves mostly absolute-scaled data.

B. Types of Software Measurement

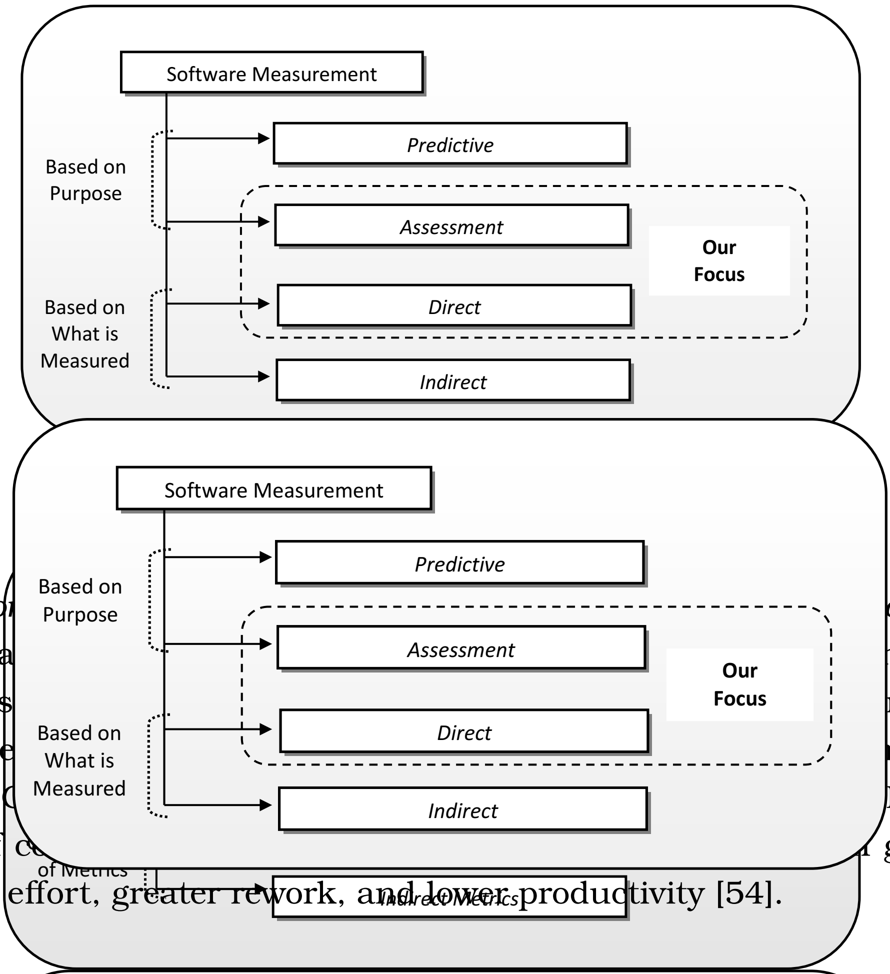
The software engineering literature [79, 81] provides us with different types of software measurements. For example, while the *purpose of measurement* leads to one classification, the *nature of the attributes measured* results in a different categorization.

Based on the purpose (or usage) of measurement, there are two different categories of measurement: *assessment* and *predictive* measurement [79]. Assessment measurement is concerned with measuring the current value of some attributes. This aims at capturing better insights into the status of an entity. On the other hand, predictive measurement focuses on formulating a mathematical model based on current status of an entity. The model assists in envisaging any desired future measure(s), and thus offers a means to manage and control a target system.

However, measurement can also be categorized by the way it is being conducted: *direct* measurement and *indirect* measurement. While the former involves measurement of only one attribute, the later entails more than one attribute [81]. For example, measuring the size of a program requires only one attribute - total number of lines of code. On the other hand, measuring productivity of associated developers demands two attributes - both number of lines and devoted effort in terms of hours spent. The productivity can only be determined by dividing total lines of code by total hours spent to develop the program.

3.3.2 Software Metrics

A software metric can be defined as *"a quantitative measure of the degree to which a software possesses a given attribute"* [112]. For example, the *number of fields* and the *number of methods* defined in a class of an object-oriented software system are two metrics that reveal the *informa-*



tion sto ality in that cla nternal aspects nal as- pects (e er val- ues of (M (i.e., lack of c greater design effort, greater rework, and lower productivity [54].

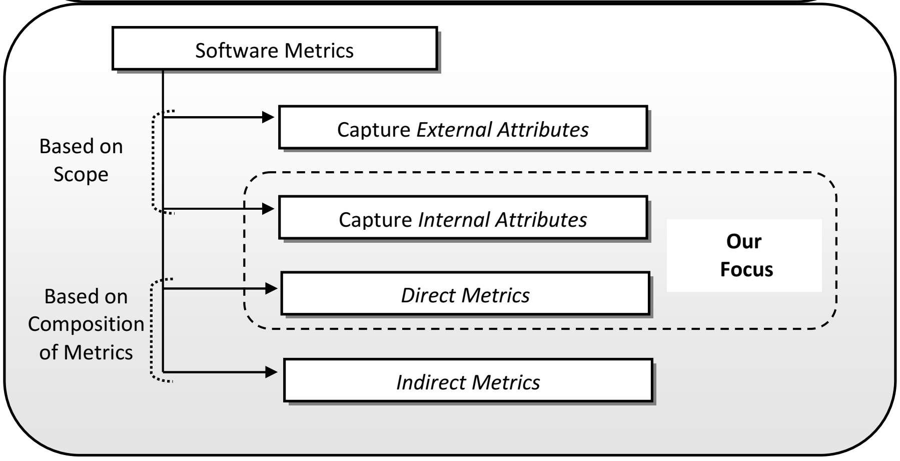

Software Metrics
Based on Scope
Capture External Attributes
Capture Internal Attributes
Our Focus
Based on Composition of Metrics
Direct Metrics
Indirect Metrics

Figure 3.4: Types of Software Metrics [81]

Consequently, a software metric can be classified into two categories based on the scope that is covered: metrics that capture internal attributes and metrics that capture external characteristics. The former, as the name implies, is specific to the entities involved, and measured by analyzing the features of its own (e.g., size, modularity, coupling). The later is involved with the surrounding environment and captures operating characteristics (e.g., maintainability, comprehensibility, and reliability). However, while the internal attributes can be determined statically, the external attributes are generally recorded dynamically (relating to the associated environment) [1].

As principle tenants of a measurement process, software metrics fall into two more categories: direct and indirect metrics [81]. A direct metrics is defined as *"a metric that does not depend upon a measure of any other attribute"* [112]. It refers to the *absolute/raw counts* of an attribute (the value of a direct metric can be any positive integer including zero values). On the other hand, *indirect metrics* are those which can be obtained by applying a mathematical function (e.g., average, summation, percentage, median) on two or more (not necessarily direct) measures. For example, *total lines of codes* in a given program is direct metric. But the *developers productivity* (obtained by dividing the total lines of code by total hours of effort devoted by the associated developers) is an indirect metric.

In our work, we define a set of direct metrics that capture the internal attributes of Java-based software systems (cf. Figure 3.4), which allow us to investigate the use of fields, properties, and inner classes in Java-based software systems. Once the desired set of metrics are defined, however, these are required to be extracted from the selected software artifacts. We now describe the basic process of defining the required software metrics and the process of collecting them.

Defining Software Metrics

According to Kitchenham et al. [123] we need to *"define all software measures fully, including the entity, attribute, unit and counting rules"*.

The scope of this study covers only the software product metrics. In particular, we compute direct metrics and internal attributes (cf. Figure 3.4). For this purpose, our metrics computation model takes a class of an object-oriented software system as an entity and the associated

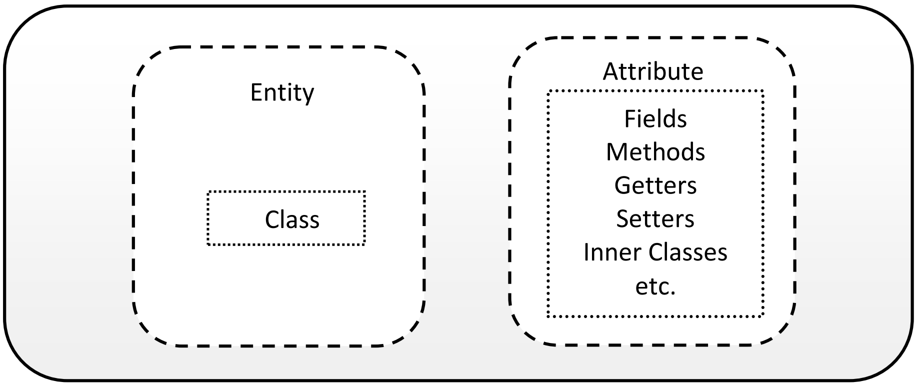


Figure 3.5: Entity and Attributes in Our Study

elements of interest as attributes (cf. Figure 3.5). For example, given a particular class, we compute the *number of getter methods* and the *number of setter methods* to gain insights into the extent of *data retrieval* and *modification* functionality, respectively, implemented in that class.

Moreover, all the computed measures covered in this study aim at capturing different aspects of investigated features of the Java programming language. We describe the definitions of these metrics in the respective chapters.

Extracting Software Metrics

A typical Java-based software system contains many different components (cf. Figure 3.6). These include compiled class files, images, sounds, and configuration files. All these components are bundled together into a single unit by archiving and compressing them into a special type of file - *Jar* (Java archive). Jar files facilitate the distribution process of the components that belong to a particular application (e.g., desktop, applet).

There are various common functionalities (e.g., graphical user interface, database transaction and connection management) that are often required in many Java applications. To facilitate application development, such functionality are usually adopted from third party libraries,

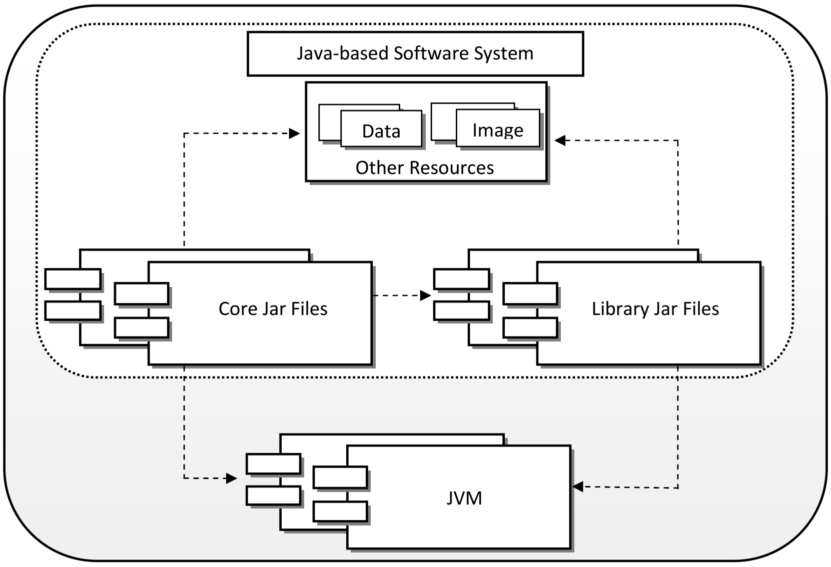


Figure 3.6: A typical Java-based software system [216].

components, plug-ins, etc. For example, Apache *Cayenne*[4] is an open source persistence framework that offers the functionality of mapping any relational database to Java objects. Apache *Commons CLI*[5] is a command line options processing library that offers necessary API for parsing and validating a command line interface. These libraries are encapsulated and deployed in jar files that can be consumed by tools (e.g., JVMs, compilers). An application that requires object relational mapping features and command line support can employ the above two utilities for these purposes, and focus on the development of only the core components that are necessary to satisfy the specific requirements of the application. Both the core components and the third party libraries (if any) - in the forms of individual building block (i.e, Jar files) - constitute the desired software system. However, we should only measure the core artifacts as they are the sole representatives of the developer decisions with respect to the software under investigation.

Given a Java-based software system, we use the associated jar files as input. Figure 3.7 depicts an overview of the metrics extraction process. To distill necessary metrics data from a software system, we used *jCT* [139] - a metrics extraction framework developed as part of this

[4] http://cayenne.apache.org/
[5] http://commons.apache.org/cli/

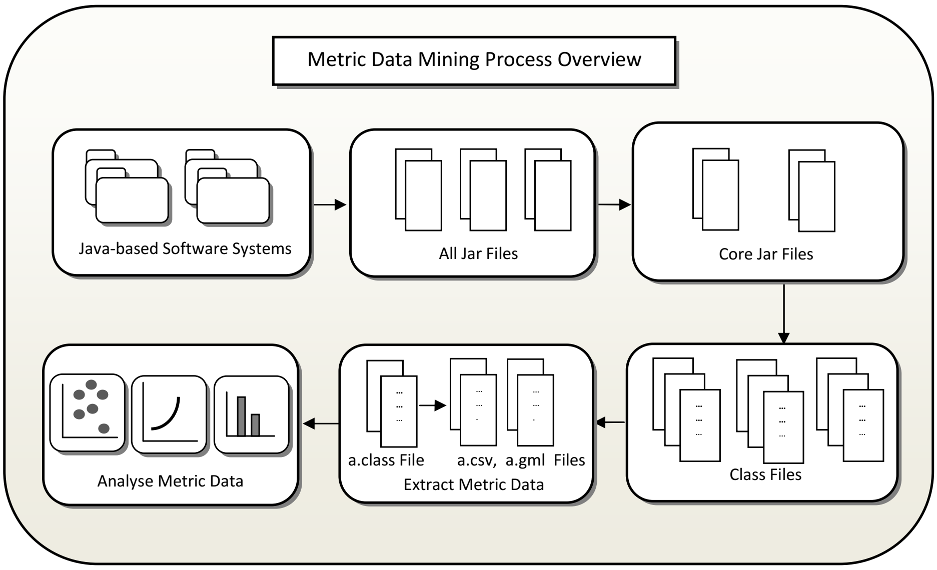


Figure 3.7: The metrics data mining process.

study.[6] jCT reads a configuration file (cf. Listing 3.1) [77] located in the root directory of every system in the corpus. This file contains meta-data authored by the creators of Qualitas Corpus [77]. jCT constructs the input data set necessary to perform metrics extraction by processing this meta-data. In particular, we use the `sourcepackages` entry to determine which classes are *core* (as defined by Qualitas Corpus [77]) and should be considered for analysis. This step is required since contemporary software systems, in general, make extensive use of third-party libraries (cf. Figure 3.6). Therefore, we need a mechanism to select the right set of artifacts for analysis [216].

Once the right set of artifacts has been identified, we use the *class file*s for desired software metrics data extraction. A class file is the compiled version of a Java source file. The Java compiler takes a source file (with .java extension) and translates it to a (set of) class file(s) (with .class extension). These files contain platform-independent bytecode instructions (cf. Figure 3.8) and metadata that contains, for example, information to locate and load classes, resolve method invocations, and enforce security constraints. Both, the bytecode instructions and the metadata, make up a runtime image of the corresponding Java class(es) that the Java VM uses to produces machine-dependent code at runtime.

[6] jCT (*Java Code Tomograph*) is presented in Appendix B

```
  /** The content of the configuration file */
fullname = Ant
domain = parsers/generators/make
notes = -
acquisitiondate = 2010-07-26
acquisitionperson = Ewan Tempero
language = Java
languageversion =  -
origin = Apache
url = http://ant.apache.org/
releasedate = 2010-05-07
opensource = true
obfuscated = false
ersionnotes = Release date from news
sourcepackages = org.apache.tools
ource = false
binaryroot = bin/apache-ant-1.8.1
exclude = etc
version = 1.8.1
qcname = ant
```

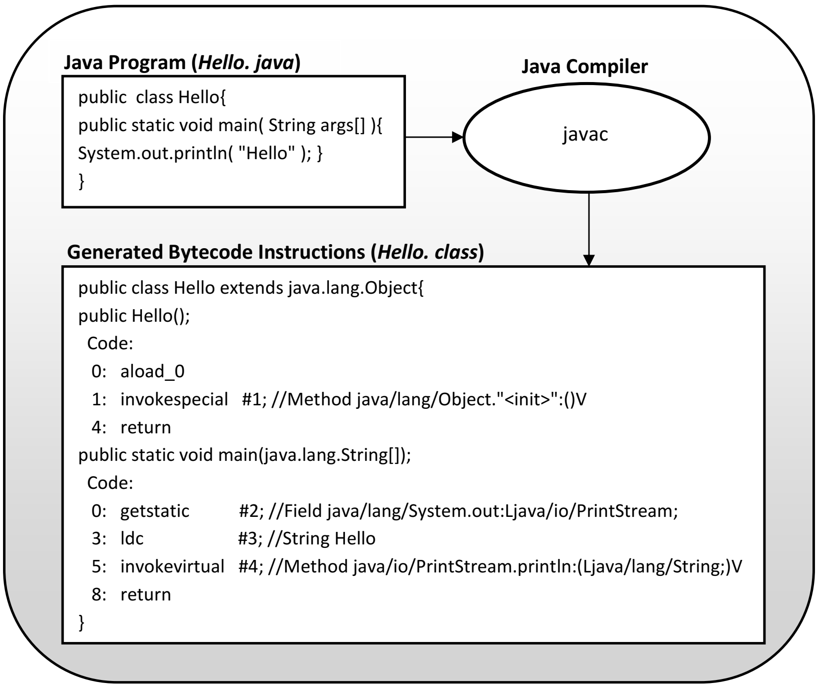
Listing 3.1: The content of the configuration file

To extract metrics data, we use the bytecode contained in the class files. To gain an insight into bytecode, consider a simple Java program that just prints *Hello* for example. When this program is compiled, the Java compiler produces a class file that contains a set of bytecode instructions shown in Figure 3.8. We process such bytecode instructions to mine necessary information for this study.

Our decision to use bytecode, rather than using source code, is motivated by the fact that except comments all the information of a source file are captured by the corresponding class file [140, 216, 217].[7] As a result, bytecode can be used in place of source code to yield the required information. In fact, a vast number of researchers (e.g., [35, 140, 144, 145, 206–209, 216]) used bytecode in their studies also.

Yet, bytecode is not exactly the replacement of source code. There are some constructs that are being eliminated as a result of the compilation process. For example, bytecode does not retain any comments

[7] A more detail discussion on class file structure and contained information is presented in Appendix B


Java Program (*Hello. java*)
Java Compiler
javac
Generated Bytecode Instructions (*Hello. class*)

```
public  class Hello{
public static void main( String args[] ){
System.out.println( "Hello" ); }
}
```

```
public class Hello extends java.lang.Object{
public Hello();
 Code:
  0:  aload_0
  1:  invokespecial   #1; //Method java/lang/Object."<init>":()V
  4:  return
public static void main(java.lang.String[]);
 Code:
  0:  getstatic        #2; //Field java/lang/System.out:Ljava/io/PrintStream;
  3:  ldc              #3; //String Hello
  5:  invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
  8:  return
}
```

Figure 3.8: A simple Java program and the generated bytecode instructions.

contained in the source code. Moreover, all the local variable names available in source code are also absent in the corresponding bytecode. Therefore, a study that requires such information may not consider bytecode as only source of information. But, as our research objective does not depend on such information, the choice of bytecode as source of required metrics data remains unaffected.

Still, there are some issues that deserve attention while working with bytecode. A potential limitation of working with bytecode is imposed by the use of a *code obfuscator*. An obfuscator, when applied, changes the content (e.g., class, method, and field names) of a compiled file in order to protect them from being reverse engineered, and thus make it difficult to reveal the original purpose of associated bytecode structure. This issue, however, is not a concern for us as the data set used in this work contains non-obfuscated code. In fact, code obfuscation would disqualify a system from being included in Qualitas Corpus [77].

Another issue is associated with the inner class processing mechanism of the Java compiler. While compiling a class that hosts inner class(es), the Java compiler emits a separate class for each of the hosting class, and hosted class(es). This results in an issue: whether all these classes should be counted separately or the compiled inner classes should be merged with their corresponding host classes. Depending on which approach is being followed, the experimental results could be substantially different.

Though some studies (e.g., [216, 220]) merge them with host classes, we consider them as separate ones. Our data mining process does not combine the inner class measures with that of their host classes. The resulting benefits of this decision is that we can study inner classes separately, and also reason about their structures.

Another issue that one should also consider while working with bytecode is the compiler generated synthetic methods and attributes if necessary. In our analysis, all such methods and attributes are ignored.

3.3.3 Analyzing Software Metrics

Software metrics data is required to be summarized in order to gain better insights into associated software systems. Aggregated software metrics facilitate data interpretation tasks and thus reveal the *state* of the intended aspects of software systems.

Software metrics can be aggregated using different methods. A method is defined as *a set of organizing principles around which empirical data is collected and analyzed* [68]. The scope of the method covers a wide variety of principles for analyzing, organizing, structuring, and interpreting all empirical data to address each aspect of the desired research problem.

Given the diversity in the field of empirical software engineering, deciding on a suitable method is still a non-trivial task. According to Easterbrook et al. [68], *"selecting a research method for empirical software engineering research is problematic because the benefits and challenges* *to using each method are not yet well cataloged"*. It is difficult to find a well accepted catalogue that covers necessary selection criteria for the available methods, associated risks, benefits, challenges, and also the appropriateness in particular problem context. Therefore, it becomes often necessary to a select problem-specific method that suits the underlying requirements of the study. Deciding on a particular method, however, demands a number of factors to be considered.

Key Issues

What are the key factors that can affect the selection of an appropriate metrics aggregation technique? The factors include not only the problem itself but also the associated scale of measurement and also the nature of the key ingredients of empirical studies: software metrics data (cf. Figure 3.9).

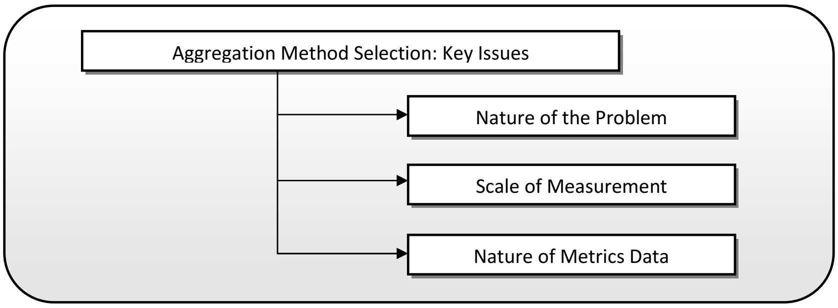
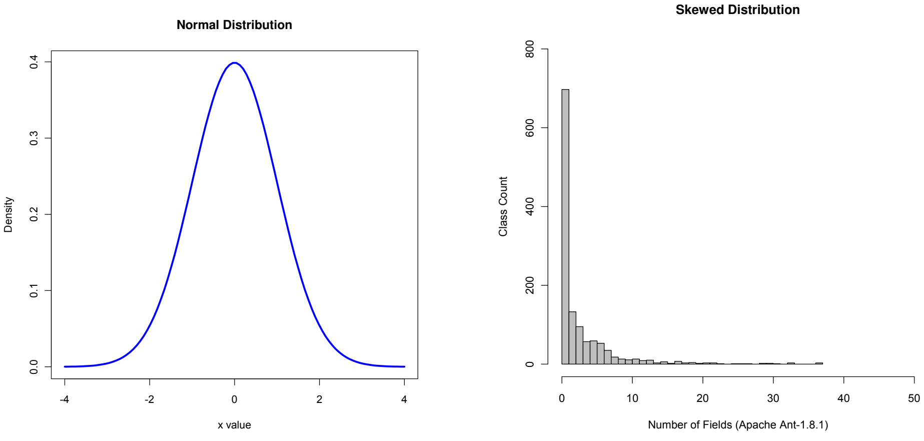


Figure 3.9: Key issues that can affect the selection of aggregation method for software metric data

The nature of our study involves an investigation of developer preferences regarding the use of language features under influence of available guidelines. This entails an analysis of the distribution of various features across different classes in Java-based software systems. For example, our focus includes whether developers create solution designs where certain proportion of classes capture most of the features (and thus become *God-like classes* - a design practice that is discouraged [183]). Therefore, the nature of our problem demands an aggregation method that can capture the underlying distribution profiles of programming language features.



(a) (b)

Figure 3.10: Distribution Pattern (a) Typical Shape of Normal Distribution (b) Observed Shape of Software Metrics (Number of Fields in Apache Ant-1.8.1) Distribution

The scale of measurement is an important issue as any aggregation technique must be aware of the underlying scale involved in the study to eliminate the chance of misleading data processing. The metrics data processing in our study does not involve any nominal, ordinal, or interval scale data but absolute-scaled data.

Software metrics data is, in general, *skewed* in nature [82, 124, 204]. It exhibits positive skewness with long tails (cf. Figure 3.10(b)). Our metrics data distributions are no exception. Therefore, the underlying aggregation technique is required to be aware of the asymmetric nature of software metrics distributions.

Available Techniques for Software Metrics Analysis

It is customary to use the commonly available central tendency measures (e.g., mean, median) to summarize metrics data. But the *non-Gaussian* nature of software metrics data makes those measures somewhat inappropriate and often misleading [217]. The reason is that these techniques are not found to take into account the underlying nature of metrics data distributions. *The more skewed the distribution, the more likely a sampled mean will underestimate the true underlying mean* [121].

Table 3.3: Different Approaches for Summarizing Software Metrics Data [216, 221]

Metric Summarizing Approaches	
Approach	**Methods/Measures include:**
Conventional	Mean Median Standard Deviation Skewness Kurtosis
Distribution Fitting	Power Law Analysis Pareto Principle
Inequality Analysis	Gini Coefficient Lorenz curve

The common standard descriptive statistical measures like “average" or “arithmetic mean" can only work if the measured population has a Gaussian distribution (cf. Figure 3.10(a)). They tend to be less-informative, and often deceptive if used in summarizing a data set that is skewed. For example, the mean value of the number of fields in classes of ant-1.8.1 (cf. Figure 3.10(b)) is 2.90 whereas the median is 1. The underlying reason for this discrepancy can be attributed to the presence of a few large values (compared to the others) that create the long tail. In ant-1.8.1, 39.88% of the classes do not comprise any fields at all but 0.88% classes contribute 9.89% of total fields. This indicates an (ecological) fallacy [172] of mean in representing the actual situation in this case.

Though there are measures available for skewed data analysis (e.g., skewness and kurtosis that capture the asymmetry and the peakedness of data, respectively), they offer unbounded values and are also affected by the underlying data set size [217]. As a result, the flexibility to compare different aspects of software metrics data straightway becomes somewhat limited.

As an alternative approach, many researchers (e.g., [35, 61, 219]) try to fit different distributions (e.g., log-normal) to software metrics data for various purposes. For example, while Baxter et al. [35] used *power laws* to understand the *shape* of Java-based software systems, Concas et al. [61] used the same for demonstrating the presence of scaling laws in a Smalltalk-based software systems. Moreover, Vasa et al. [219] used a power scaling relationship to analyze the recurring structural and evolutionary patterns in object-oriented software systems.

Unfortunately, the distribution fitting techniques cannot be used for meaningful comparison of software metrics data associated with different software systems [216]. Moreover, some of these techniques (e.g., *Pareto* principle, *Log normal*, and *Power law* distribution [181]) do not allow any zero or negative values. But software metrics data often contains substantial number of zero values [92]. Therefore, the application of these distributions becomes restricted or incur additional work (i.e., eliminating zero values or transforming them to acceptable forms).

There exist, however, a promising alternative for analyzing software metrics data with econometric techniques (e.g., inequality measures like Lorenz curve [135] or the Gini coefficient [89],). Many researchers (e.g., [140, 216, 217, 221, 222]) used such techniques in their studies for different purposes. For example, Vasa [216] applied the Gini coefficient for summarizing software metrics data to study software evolution, whereas Lumpe et al. [140] adopted the same technique to analyze the property mechanism in Java-based software systems.

Our Focus

Our study involves a collection of software artifacts that varies in terms of their functionalities, underlying domains, and size. Therefore, a method must facilitate meaningful and direct comparisons of software metrics data across different software systems regardless of their underlying attributes (e.g., size, distribution). In our study, we use an inequality measure (i.e., the Gini coefficient) to summarize software metrics data.

3.3.4 Inequality-based Software Analysis

In 1905, Lorenz [135] introduced a graphical representation to study the distribution of a particular attribute in a given population. In fact, this technique aims at measuring the extent of *inequality in income distribution* in a society.

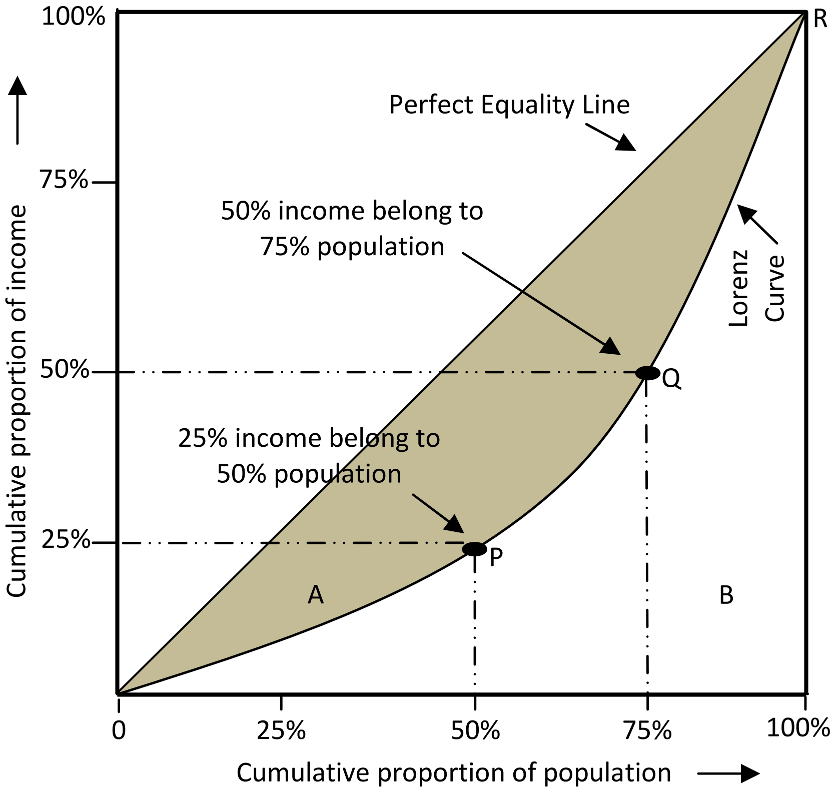
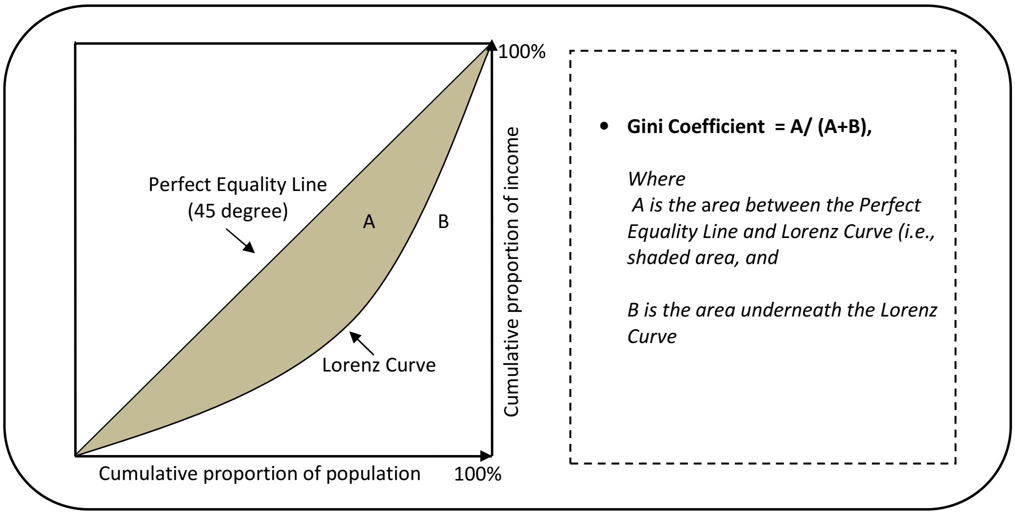


Figure 3.11: Lorenz Curve Illustration

Consider, for example, a society with population of size n and associated income of w. To graphically represent the underlying inequality of income distribution among the members of the society, the *cumulative proportions of population* are plotted along the x-axis, and the corresponding *cumulative proportions of income* are plotted along the y-axis. The resulting curve is known as the *Lorenz curve* (cf. Figure 3.11).

Each point (x, y) on the Lorenz curve captures the relationship that y% income is received by the bottom x% of the population. For example, while the point P on the Lorenz curve depicted in Figure 3.11 indicates that 50% of the total population earn only 25% of the total income, the other point Q denotes that 75% of the population earn 50% of the total income. The rest (50%) of the income belong to the remaining 25% of the population. This way the Lorenz curve reaches point R that implies 100% of the population own 100% of the total income.


100%
Perfect Equality Line
(45 degree)
A
B
Lorenz Curve
Cumulative proportion of income
Cumulative proportion of population
100%

- **Gini Coefficient = A/ (A+B),**

Where
A is the area between the Perfect Equality Line and Lorenz Curve (i.e., shaded area, and

B is the area underneath the Lorenz Curve

Figure 3.12: Lorenz Curve and Gini Coefficient

If the income happens to be equality distributed (i.e., everyone's income is exactly same), then the Lorenz curve produces a straight diagonal line – *perfect equality*. On the other hand, any inequality in distribution makes the Lorenz curve bend in the middle (cf. Figure 3.11). This works with non-negative data, and so the Lorenz curve lies between the line of *perfect equality* and that of *perfect inequality* (i.e., no one earns except one person).

The Lorenz curve captures inequality in distribution graphically, it is, however, more practical (and often more desirable) to work with numerical values. Hence, we need a numerical value that can capture the graphical representation. This is the Gini coefficient which is defined as a ratio of the areas on the Lorenz curve diagram. If the area between the 45° line of perfect equality and the Lorenz curve is A, and the area under the Lorenz curve is B, then the Gini coefficient is $A/(A+B)$ [234] (cf. Figure 3.12).

More precisely, for a given population of metrics data x_i, $i = 1$ to n, that is indexed in an increasing order such that $x_i \leq x_{i+1}$, the Gini Coefficient G is defined as:

$$G = \frac{1}{n}\left(n+1-2\left(\frac{\sum_{i=1}^{n}(n+1-i)x_i}{\sum_{i=1}^{n}x_i}\right)\right) \tag{3.3.1}$$

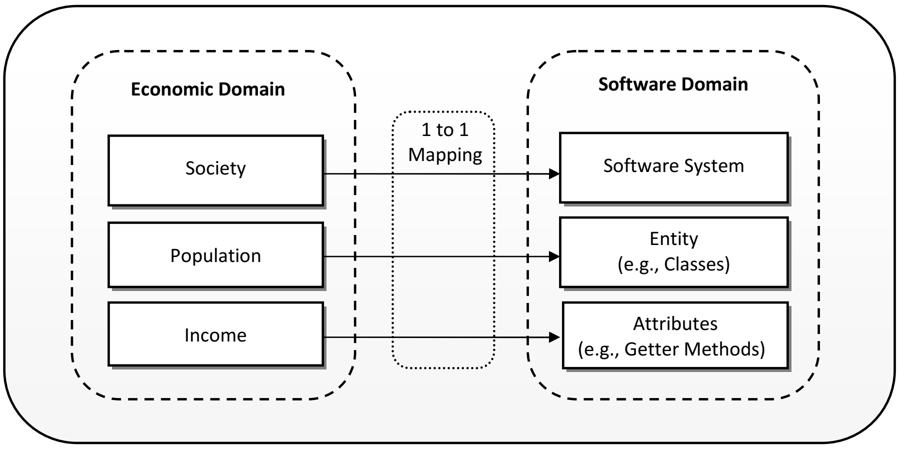


Figure 3.13: Applying Inequality Measure in Software Analysis

The Gini coefficient is a number in the interval [0,1), where 0 denotes *perfect equality* (i.e., the allocation of features adheres to an even pattern) and 1 stands for *perfect inequality* (i.e., there is only one *God*-class in the system). The higher the value of Gini coefficient, the more concentrated the distribution is.

Application in Software Analysis

How does econometric measurement work in the context of software analysis? In order to analyze software systems with an econometric measure, we have to map the associated entities from the economic domain to the software domain as illustrated in Figure 3.13. Given a software system, we consider it as a society. The classes and associated attributes are regarded as population and income, respectively.

For example, if we are interested in revealing the distribution pattern of getter methods in netbeans-6.9.1, then we can use the corresponding Lorenz curve (cf. Figure 3.14). We see that the getter methods in netbeans are concentrated in only a few classes. About 78.76% classes of netbeans do not define any getter methods at all. The corresponding Gini coefficient is 0.83, indicating a very high concentration of getter methods in netbeans-6.9.1.

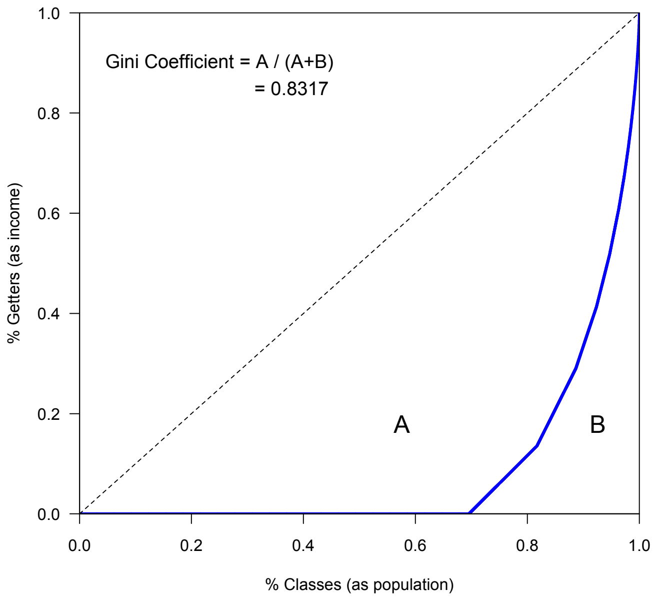


Figure 3.14: Lorenz curves example - Getter methods hosting profile of classes in Netbeans - 6.9.1

Key Benefits

Table 3.4: The Benefits offered by Inequality Measure

Gini Coefficient - Fast Facts		
Feature	**Attribute**	**Benefit(s)**
Coefficient Value	[0,1)	A bounded value offers direct means to compare different software metrics data (unlike average that produces different values depending on the size of the associated data set). This provides us with the opportunity to compare desired aspects of software systems straightway.
Data Distribution	Agnostic	As underlying data distributions do not affect the Gini coefficient, the same way as central tendency measures it allows us to compare different software metrics data conforming to different distributions. For example, it can be applied conveniently to process software metrics data that is skewed.
Data Set Size	Agnostic	Different software systems can comprise a varying number of artifacts. The Gini coefficient is, however, a size independent measure. It captures the effective inequality in distribution irrespective of the size of the population.

Using the Gini coefficient in software metrics data analysis provides us with a number of benefits. These include the *bounded value* and independence of *data distribution* and data set size. These key benefits are described in Table 3.4.

Moreover, the Gini coefficient can often reveal valuable insights into the change in distribution of certain aspects of software systems which remains completely blind to traditional central tendency measures (e.g., median) [217].

3.4 Summary

In this chapter, we described the methodology adopted in this work. The nature of our study motivated us to select an empirical research approach that involves collecting quantitative information, and summarizing the distilled data with suitable aggregation techniques.

We described different characteristics (e.g., size distribution) of our data set. Moreover, we classified the software systems in Qualitas Corpus into 12 different categories (e.g., database, middleware, tools). Though this classification is based on the nature of functionality provided by the associated software systems, we do not claim it to be the only one. But it serves our purpose (i.e., ensures diversity of the data set, and also forms the basis for domain-specific characterization of developer preferences in using programming language features in Java-based software systems).

In addition, we discussed our measurement procedure for the selected software systems (i.e., Qualitas Corpus). The description covered two different aspects: (i) basic measurement process and (ii) underlying software metrics.

In the former aspect, we illustrated basic measurement principles, and associated scales of measurement that govern the application of appropriate summarizing techniques of software metrics data. We described scale-specific statistical operations that are employed during software measurement. We argued that the nature of metrics data we work with conforms to the absolute scale.

In the later aspect, we discussed software metrics extraction, processing, and aggregation techniques. We illustrated how we extracted desired software metrics data from Java-based software systems. We justified our choice of working with bytecode rather than source code. In addition, we discussed available software metrics aggregation techniques, and inequality measures - a recently adopted technique for software metrics summarization. We described the usefulness of this technique in the context of our work.

There are, however, some limitations in our methodology. For example, we used open source Java-based software systems to base our results on. Therefore, we do not know whether or not the resulting conclusions would remain same in case of other programming languages or software systems that are not being developed under open source development model. Another issue could be the representativeness of the data set. But this concern is curtailed by the diversity (cf. Table 3.2) of software artifacts in Qualitas Corpus [77]. This raises our confidence regarding the external validity of the conclusions arising from this study within the context of Java-based software systems.

Chapter 4

Field Analysis

In this chapter, we focus on understanding developer behavior in using fields. We begin with setting the stage for studying the use of fields in Java-based software systems and formulating specific research questions. We then describe the analysis method (that comprises a definition of a set of software metrics and description of metrics analysis approach) in Section 4.2. We present the results of our investigation as observations in Section 4.3. Finally, we summarize the key findings of this chapters in Section 4.4.

4.1 Introduction

A fundamental principle of object-oriented programming is *data encapsulation* [83, 157]. It aims at ensuring the integrity of the state of object instances by concealing them from the surrounding environment. To facilitate controlled access to an object's state, developers are provided with a key mechanism - visibility modifiers (e.g., private, public) to restrict the access to fields that represent an object's state.

Java provides four visibility modifiers for fields: *public, protected, private*, and *package* [26]. Each of them regulates varying level of access to the associated field. A field with the public visibility modifier can be accessed from anywhere in the software system. While the protected modifier restricts the scope of field access to the same package and subclasses, the package modifier limits the scope to the associated package only. The private modifier limits it even further, enforcing the access to be scoped to the defining class only. Though we have varying level of visibility control, general recommendations in software engineering theory limit public access [183].

But how do the developers use fields in Java? Do they adhere to the recommendations (e.g., *don't expose state if you don't have to* [12])? In a recent study, Tempero [207] analyzed the *use of fields*, and discovered that most studied systems contain non-private fields, which may be taken as an indicator for a systematic breach of data encapsulation in those systems. However, the actual number of exploits is much smaller. Only 12% of the *exposed classes* (i.e., classes that contain at least one non-private field [207]) are subject to non-private field access. Tempero stipulated that non-private fields may be a result of accidents (or oversights) rather than conscious design decisions as the studied systems do not take advantage of non-private fields.

In this chapter, we investigate developer tendencies in adhering to available advices regarding the use of fields in particular, and the associated design choices in general. We revisit some of the results (e.g., extent of exposed classes) of the above study where appropriate. Thus extend its validity by replication - an important element in empirical studies emphasized by many researchers (e.g., [43, 194, 233]). Moreover, following Tempero's recommendation [207], we restrict our focus on mutable (e.g., non-final) and object-based (i.e., non-static) fields. While the final fields are considered to be “safe" from the field exposure perspective [207], static fields tend to be used in different roles (i.e., used to define constants which are not object-specific) than instance fields.

In particular, we address the following questions:

- **RQ1**: What is the typical field distribution profile that developers usually practice? Do they follow available recommendations?

- **RQ2**: Does field distribution vary across different domains (e.g., middleware, database)? Do the results of RQ1 hold at domain level?

- **RQ3**: Do developers define fields in all classes in Java-based software systems? What is the typical distribution of field hosting classes and field inheriting classes?

- **RQ4**: Given a software system, are the volume and distribution of its fields correlated?

- **RQ5**: What is the typical profile of field exposure in Java-based software systems? Do developers confine exposed fields in a few classes or do they disperse them in almost every field hosting class?

4.2 Analysis Method

To answer the above research questions, we defined a set of software metrics, and analyzed them using different measures (e.g., the Gini coefficient). In this section, we describe them in more detail.

Definitions of Metrics

To investigate the field distribution profiles in studied Java-based software systems, we collected different fields measures (cf. Table 4.1). These measures capture different fields with various visibility modifiers (e.g., public, private). We also defined a set of class measures that assist us in studying the distribution of different aspects of classes with fields (e.g., how many classes typically comprise fields, how many of them host exposed fields, how many of them access exposed fields).

Data Analysis Approach

The analysis of the above software metrics data is descriptive in nature. We used both mathematical (e.g., the Gini coefficient) and several graphical measures: (i) histograms to reveal the frequency distribution,

Table 4.1: The Collected Measures.

Field Measures		
Name	**Purpose**	**Relation**
Number of Attributes (NOA)	Counts all defined fields in a class	-
Number of Public Attributes (NOPubA)	Counts all public fields in a class	NOPubA ⊆ NOA
Number of Protected Attributes (NOProA)	Counts all protected fields in a class	NOProA ⊆ NOA
Number of Private Attributes (NOPriA)	Counts all private fields in a class	NOPriA ⊆ NOA
Number of Package Attributes (NODefA)	Counts all fields with default or package visibility in a class	NODefA ⊆ NOA
Number of Exposed Attributes (NOExpA)	Counts all fields with non-private visibility in a class	NOExpA ⊆ NOA

Class Measures		
Name	**Purpose**	**Relation**
Number of Classes with Attribute(s) (NOCA)	Counts number of classes containing at least one defined field	-
Number of Classes with Public Attribute(s) (NOCPubA)	Counts number of classes with at least one public field	NOCPubA ⊆ NOCA
Number of Classes with Protected Attribute(s) (NOCProA)	Counts number of classes with at least one protected field	NOCProA ⊆ NOCA
Number of Classes with Private Attribute(s) (NOCPriA)	Counts number of classes with at least one Private field	NOCPriA ⊆ NOCA
Number of Classes with Default Attribute(s) (NOCDefA)	Counts number of classes with at least one Default field	NOCDefA ⊆ NOCA
Number of Classes with Inherited Attribute(s) (NOCIA)	Counts number of classes with at least one inherited field	NOCIA ⊆ NOCA
Number of Classes with Exposed Attribute(s) (NOCExpA)	Counts number of classes with at least one Exposed field	NOCExpA ⊆ NOCA
Number of Classes with Mostly Exposed Attribute(s) (NOCMExpA)	Counts number of classes with Exposed Fields where more than 50% of Total Fields (NOA) are Exposed (NOExpA)	NOCMExpA ⊆ NOCExpA
Number of Classes being Indirectly Accessed (NOCBIA)	Counts number of classes whose fields are accessed indirectly from other classes	NOCBIA ⊆ NOCA
Number of Classes who Indirectly Access other's Attribute(s) (NOCWIA)	Counts number of classes which access fields defined in other classes	NOCWIA ⊆ NOCA

(ii) box plots to capture summarized views, and (iii) the Lorenz curve to depict inequality in distributions.

To understand field distribution profiles (i.e., distribution of fields with different visibility modifiers, and distribution of fields across different domains - RQ1 and RQ2) in Java-based software systems, we relied on frequency distribution analysis - a technique used by many researchers (e.g., [49, 207–209, 225]) for conducting similar studies. We present the resulting outcome at both Qualitas Corpus level (using box plots) and individual system level (using bar plots). Based on the frequency of different measures regarding various fields (e.g., private fields, public fields), we answer whether developer practices comply with the recommendations they are provided with.

To gain an insight into the distribution of field hosting classes (RQ3), we used an inequality measure (the Gini coefficient). Given a software system, its private fields, for example, are treated as *wealth* and classes[1] as *populations*. We computed the Gini coefficients of fields with different visibility modifiers, and summarized them using box plots to yield an overview of the nature of distribution.

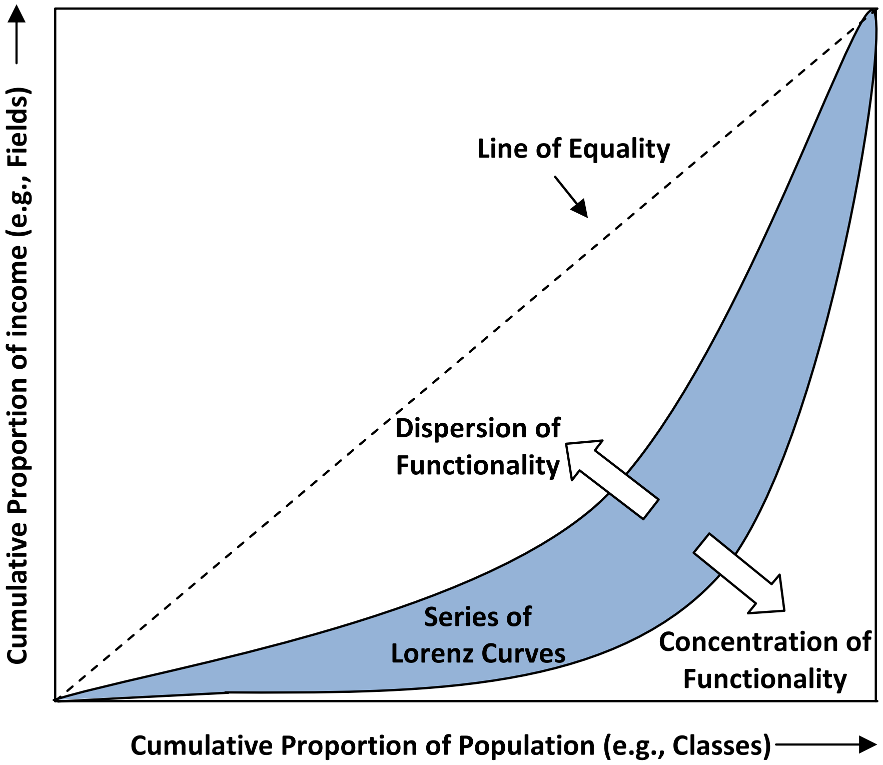

Cumulative Proportion of income (e.g., Fields)
Line of Equality
Dispersion of Functionality
Series of Lorenz Curves
Concentration of Functionality
Cumulative Proportion of Population (e.g., Classes)

Figure 4.1: Region of Feature Distribution.

In fact, the Gini coefficient of a particular measure is not always the same for all systems [217]. The values vary depending on the system's specific organization of functionality. To yield an insight into the range of Gini coefficients of that measure for each of the analyzed software systems, we plotted the corresponding Lorenz curves for all of them. As a result, we obtained a pattern like the one depicted in Figure 4.1. We used this pattern to symbolically indicate any typical bounded region of functionality distribution or the “decision frame" for the measured feature [210].

[1] Fields in interfaces are implicitly public, static and final [94]. As we do not count static fields, fields in interfaces are excluded.

Such bounded region can provide us with an indication of consistency in developers in employing associated functionality (as represented by the given measure). We can attribute such consistency to an accepted practice in Java-based software development. To quantify the bounded region, we computed Inter-Quartile range of the Gini coefficient values for given measures to identify any typical boundaries of that metric for the software systems in the Qualitas Corpus.

In addition, to understand whether volume and distribution of fields are related (RQ4), we used proportion (as representative of volume) and the Gini coefficient (as representative of distribution) of fields in the software systems of the Qualitas Corpus. We checked for any linear relationship between the computed proportions and Gini coefficients. A negative correlation would imply that developers disperse fields when volume increases, and vice versa.

To investigate the field exposure profiles (RQ5), we analyzed the frequency distribution of different non-private fields. In addition, we used a *normality test* as a secondary check of inequality. For this purpose, we used the *Shapiro-Wilk* test for normality [189] to a variety of promising measures in order to identify possible Gaussian distributions. We either accepted or rejected the *NULL*-hypothesis for normality of selected measures. Besides, we also built a linear model to identify any relation between a pair of measures (e.g., classes that host exposed fields and classes that access those exposed fields).

4.3 Observations

In this section, we present the outcomes of our empirical investigation of different aspects of the usage patterns of fields in Java-based software systems. This section is divided into the following subsections: (i) field distribution (answers RQ1 and RQ2), (ii) field hosting classes (answers RQ3), (iii) relation between proportion and distribution of fields (answer RQ4), and (iv) field exposure profiles in Java-based software systems (answers RQ5).

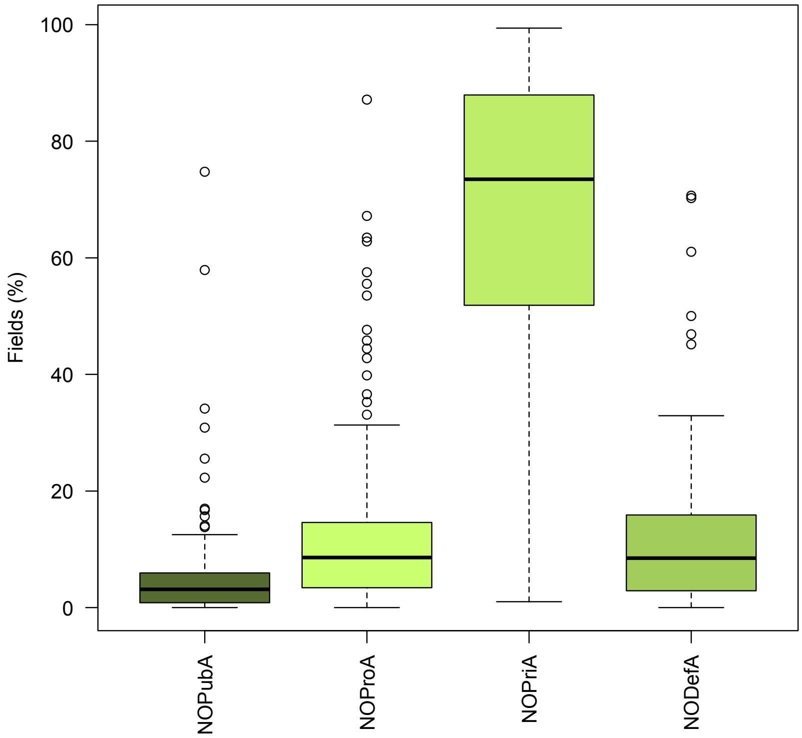


Figure 4.2: Boxplot of field distributions by modifier in the Qualitas Corpus.

4.3.1 Field Distribution

We present the results of our investigation by dividing them into: (i) distribution of fields with different visibility modifiers, (ii) distribution of fields across different domains.

Distribution of Fields with Different Visibility Modifiers

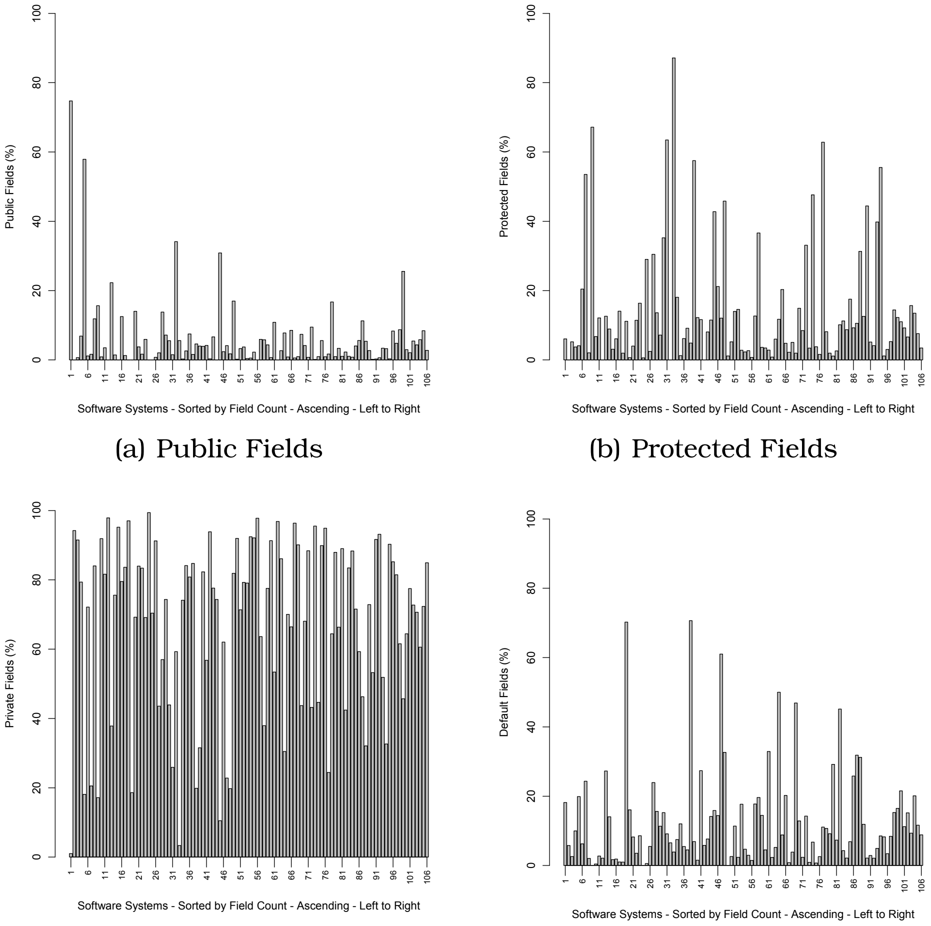
Public Fields

The use of public fields is very limited in the software systems in the Qualitas Corpus (cf. Figure 4.2 and Figure 4.3(a)). They comprise varying degree of fields. We found 10 software systems (i.e., checkstyle, displaytag, informa, jFin_DateMath, jmoney, jpf, mvnforum, rssowl, tapestry, trove) in the Qualitas Corpus that employ no public fields at all. Among the rest of the software systems, 73% comprise less than 5% public fields (and 61% have only less than 2%).

The observed proportions of public fields suggest that developers do not breach the object's state through public fields significantly. It also indicates that even when developers expose object state through public fields, they do it in few cases only. The underlying reason of such limited use of public fields can be attributed to a developer tendency in adhering to available advices (e.g., [12, 183]) that discourage the use of public fields. Following the advices, developers tend to avoid the overhead associated with public fields. For example, if a class that hosts a public field changes over time (as a result of software evolution, maintenance, etc.), corresponding client classes of that public field may also require modification. Thus, the use of the public field may result in a series of changes (aka, ripple effect). Avoiding public fields, developers assist in minimizing such potential ripple effect - a criterion, required for software systems to be stable, that should be obtained during design time [185].

However, though we observed a limited practice of using public fields in most of the software systems, there are some exceptions. We found 6 software systems in the Qualitas Corpus where the proportion of public fields is quite high (more than 20%). These are fitjava (75%), jasml (58%), freecs (34%), antlr (31%), aspectj (26%), and javacc (22%).

Most of the software systems that exhibit high proportion of public fields are rather small in size. For example, fitjava comprises only 60 classes, and jasml has only 49 classes. Fitjava is a framework for integrated testing that supports collaboration among the customers, testers, and developers. It contains less than 100 fields that are distributed in 54% of its classes. A formatting (HTML) class alone contains 26% of the public fields. Jasml is a tool that allows Java class files to be decompiled, viewed, and edited through asm-like Java macro instructions. It has 171 fields that are distributed in 86% of its classes. In jasml, two data holder classes contribute 30% of the public fields. Freecs is a chat server that comprises 147 classes only. A server class alone contains 54% of the public fields. However, these software systems are exceptions, and may not be taken as a representative example of a developer's inclination towards employing high proportions of public fields in Java-based software systems in general.



(a) Public Fields

(b) Protected Fields

(c) Private Fields

(d) Default Fields

Figure 4.3: Field distribution by visibility modifier of the software systems in the Qualitas Corpus. (X-axis is sorted to indicate the inconsistency between number of total fields and fields with different visibility modifiers.)

Protected Fields

Like public fields, most of the software systems in Qualitas Corpus make use of limited protected fields (cf. Figure 4.2 and Figure 4.3(b)). About 55% of software systems define less than 10% protected fields (64% of which contain less than 5% only). Some software systems, on the other hand, comprise a higher proportion of protected fields too. About 20% of the software systems define more than 20% protected fields, and some of them contain a substantially large proportion of protected fields. We found 7 such systems that define more than 50% protected fields. These are jgraph (87%), nekohtml (67%), colt (63%), cayenne (63%), drawswf (58%), weka (56%), and webmail (54%).

Private Fields

Not surprisingly, private fields are used the most (cf. Figure 4.2 and Figure 4.3(c)). The median value of the proportion of private fields is 74%. We found more than 80% private fields in 40% software systems in Qualitas Corpus. Some of them (including informa, rssowl, jpf, tapestry, jfreechart, myfaces_core, displaytag) comprise more than 95% private fields. The highest proportion is observed in checkstyle (99%). However, these software systems have no public fields at all except jfreechart and myfaces_core that comprise the least number of public fields (less than 1%).

While private fields are extensively used in almost all the software systems in the Qualitas Corpus, only two systems (i.e., fitjava, jgraph) stand as exceptions. They comprise only less than 5% private fields. While the developers of fitjava make extensive use of public fields (75%), the developers of jgraph rely on protected fields substantially (87%). These cases, however, are not enough to represent common practice, and hence we confirm that developers usually conceal data by using the private visibility modifier.

Default Fields

The use of fields with default (package) visibility is also limited (cf. Figure 4.2 and Figure 4.3(d)). Among the software systems in the Qualitas Corpus, 82% use less than 20% default fields. There is no default field in only three systems (i.e., checkstyle, nekohtml, and proguard). We found, on the other hand, three systems with more than 50% default fields. These are jmoney, joggplayer, and pooka that have 70%, 71%, 61% of default fields, respectively.

The underlying reason of such high use of default fields can be attributed to the presence of GUI classes. In jmoney, almost 90% of the fields with package visibility are declared in less than 10% classes that implement GUI functionality. In joggplayer, less than 10% classes that provide gui and psychoacoustic setup (e.g., bitrate, stream) functionality comprise more than 50% of the default fields, resulting in a high inequality in the distribution of fields with package visibility. Like jmoney and joggplayer, high use of default fields in pooka can also be attributed to the GUI classes. We attribute the high degree of default fields to developer coding style or design choice. We can eliminate GUI builders, like netbeans, as these tools generally add a visibility modifier (e.g., private) by default.[2]

Field Distribution Across Different Domains

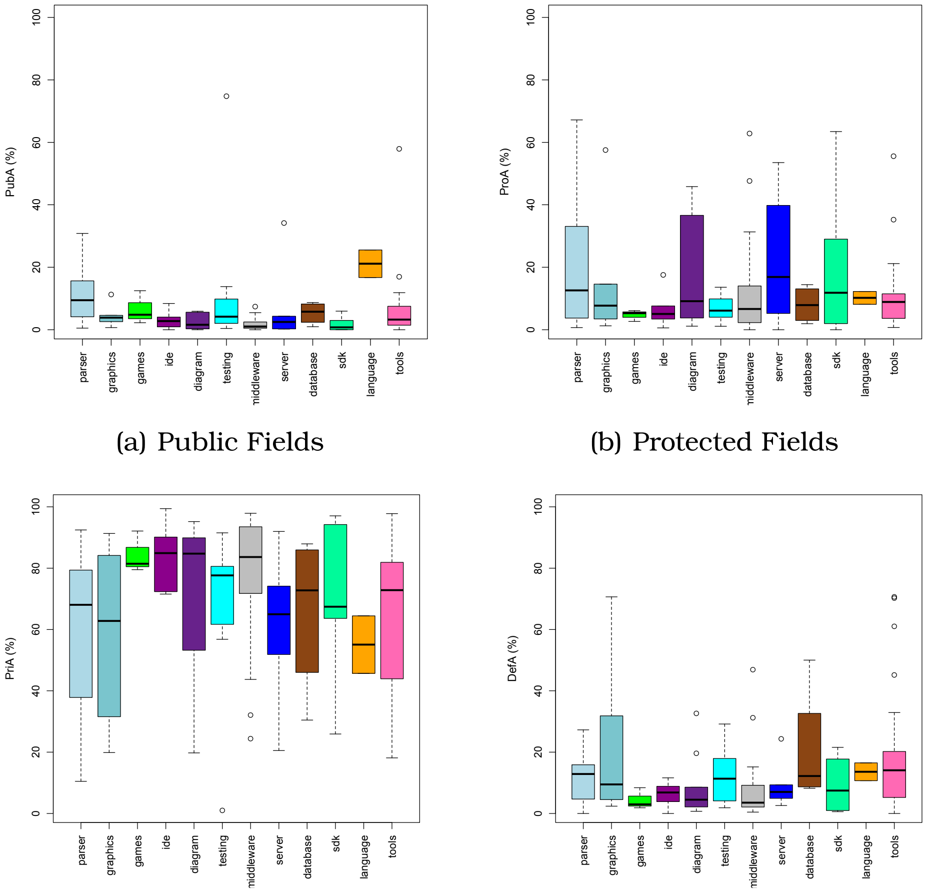
Does the use of fields vary across domains (e.g., database, games, middleware)? To facilitate our understanding on fields usage in different domains, we investigated the domain-specific proportions of fields with four visibility modifiers. The varying degree of proportions is depicted in Figure 4.4.

We observed that the use of public fields is limited in all of the studied domains. One software system (i.e., fitjava) in the testing domain stands out. Though this software system comprises 75% of public fields, the total number of fields in this system are less than a hundred. Two software systems (freecs, jasml) in the tools domain have public field proportion of 34% and 58%, respectively. Comparatively, programming languages make a higher use of public fields than systems in other domains. However, there are only two examples in Qualitas Corpus that prevents us from making any general conclusion about the nature of public fields in Java-based programming language implementations.

The protected fields have comparatively more usage in the parser, diagram, server, and sdk domains. But there are some outliers in several domains. For example, drawswf in the graphics domain is an outlier, comprising 58% protected fields. The application jgraph, webmail and weka in the tools domain employs 87%, 54% and 56% protected fields. The application cayenne of the middleware domain comprises 63% protected fields.

While private fields are substantially used in almost all the studied domains, the only exception is the application fitjava of the testing domain that has very few private fields. Default fields have more usage in the

[2] We tested this feature with the Netbeans IDE.



(a) Public Fields

(b) Protected Fields

(c) Private Fields

(d) Default Fields

Figure 4.4: Comparative proportions of different fields in various domains of the software systems in the Qualitas Corpus.

graphics and database domains. The outliers include hsqldb (50%) of database domain, jmoney (70%) and joggplayer (71%) of tools domain.

However, even though the use of fields varies across different domains, it is evident that developers tend to comply with the associated recommendations (e.g., [12, 162, 183]) that discourage to use public fields. Even a possible presence of machine generated code does not overshadow this observation (e.g., antlr code contained in checkstyle does not affect the distribution of public fields). Developers appear to exhibit a domain-agnostic inclination towards information hiding. More precisely, developers follow recommendations, only rarely do developers violate the information hiding principle due to specific domain pressures.

4.3.2 Field Hosting Classes

We studied the extent of fields with different visibility modifiers (i.e., public, protected, private, default) in Java-based software systems. But it does not inform us to what extent the classes comprise fields. In this section, we describe the results of our investigation on the distribution of field hosting classes. We present the resulting outcomes under two different facets: (i) Extent of concentration of fields across different classes, (ii) Distribution of field hosting classes in software systems of Qualitas Corpus.

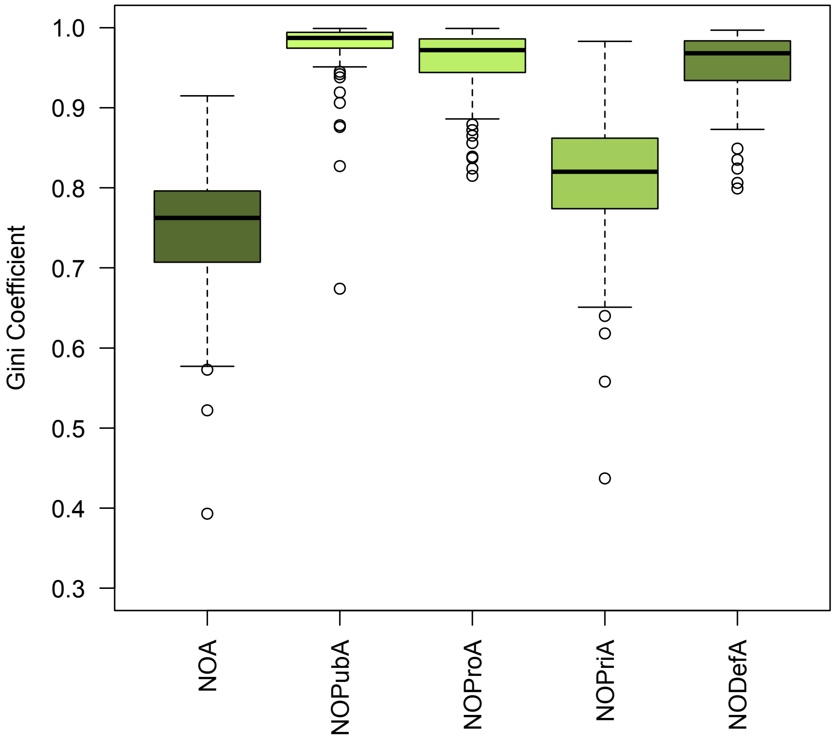


Figure 4.5: Boxplot of Gini Coefficient of Field Measures of the software systems in the Qualitas Corpus.

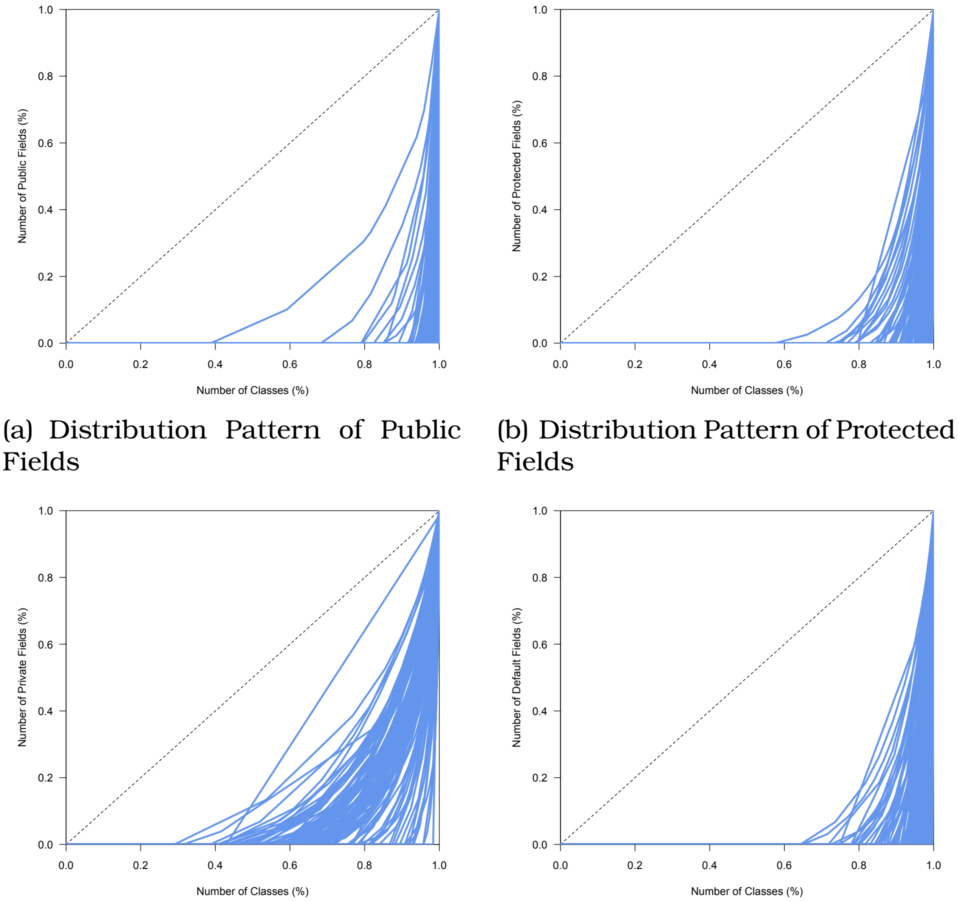
Extent of Concentration of Fields across Different Classes

The results (depicted in Figure 4.5) show that the Gini coefficients of various field measures are very high. A high value of the Gini coefficient for a given measure implies concentrated distribution of the measured entity. Therefore, the above findings indicate that fields with different visibility modifiers are highly concentrated in the studied software systems. This suggests that developers prefer to centralize data storage (as represented by fields) in typical Java-based software systems.

Based on our data set, we can empirically quantify the *width* of a such bounded region and determine typical *min* and *max* values for each measure by using the *interquartile range* (IQR). Q_1 and Q_3, the lower and upper quartile, capture 50% of the data and yield a good estimate for the width of the bounded region. For example, we can establish the width of the typical region of *Number of Attributes* by taking $Q_1 = 0.707$ and $Q_3 = 0.796$ as $[0.707, 0.796]$ with width 0.089. This interquartile range (IQR) indicates that developers usually concentrate fields distribution. We observed that fields with public, protected and default visibility modifiers are highly concentrated. Whenever they are used, developers keep them centralized in only a few classes (cf. Figure 4.6). On the other hand, fields with private visibility modifier are comparatively dispersed (cf. Figure 4.6(c)) across different classes in Java-based software systems.

There are, however, a few outliers (e.g., trove, jparse and jrat) that show comparatively more equal distribution of fields. For example, trove - a lightweight implementation of the Java collections API, exhibits the least concentration of fields (Gini coefficient = 0.393). A closer look at trove revealed that it comprises *wrapper-like* classes that implement each functionality independently, causing fields to occur in every corresponding class. As a result, a more equal distribution of fields is observed in trove.

To quantify the range of distribution of different fields, we recorded their Gini coefficients (cf. Table 4.2), and computed their interquartile range (IQR). We found that the IQR-width of private fields is substantially wider than the non-private ones. On the other hand, the IQR-width of public fields is the least one. Thus, the IQR-widths of different field distributions in the software systems of the Qualitas Corpus exhibit a relation like *IQR-width (private fields) > IQR-width (protected fields) > IQR-width (package fields) > IQR-width (public fields).*



(a) Distribution Pattern of Public Fields

(b) Distribution Pattern of Protected Fields

(c) Distribution Pattern of Private-Fields

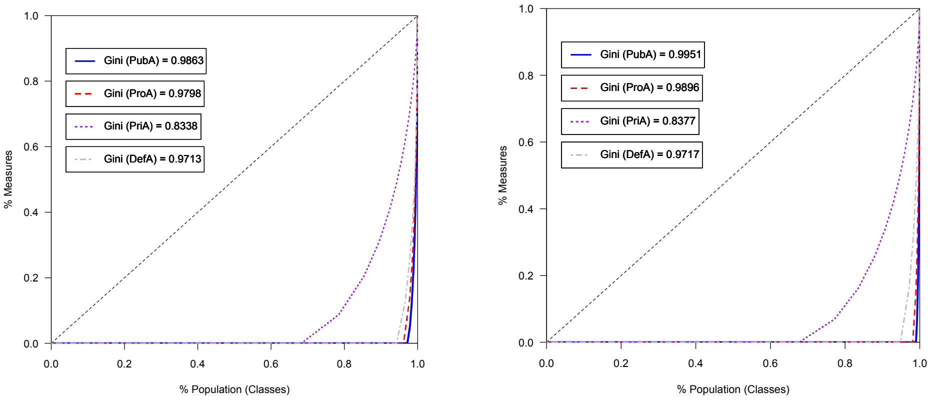
(d) Distribution Pattern of Default-Fields

Figure 4.6: Distribution patterns of fields with different visibilities - It suggests that the private fields have wider distribution than the fields with other three visibilities

Measure	**Range**	**IQR-Interval**	**IQR-Width**
Number of Public Attributes	[0.674 – 0.998]	[0.974, 0.994]	0.020
Number of Protected Attributes	[0.815 – 0.998]	[0.945, 0.999]	0.054
Number of Private Attributes	[0.437 – 0.983]	[0.774, 0.862]	0.088
Number of Package Attributes	[0.799 – 0.997]	[0.934, 0.983]	0.049

Table 4.2: Narrow bounded region of fields with different modifiers.

Given such high concentration of non-private fields, does this observation hold for large software systems like eclipse and netbeans? To answer this question, we investigated their field distribution profiles (cf. Figure 4.7). We observed that both eclipse and netbeans share almost the same distribution profile, which suggests that even in the large soft-



(a) Field Distribution in Eclipse. (b) Field Distribution in Netbeans.

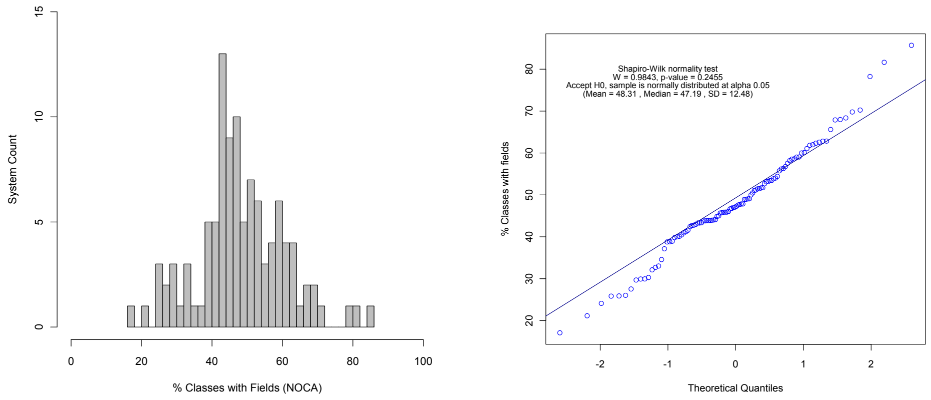
Figure 4.7: Field Distribution in two large software systems : Eclipse and Netbeans

ware systems (comprising more than 30,000 classes), private fields are dispersed comparatively more than non-private ones (i.e., public, protected and default) that are, in general, highly concentrated. In addition, the Gini coefficients of different type of fields (e.g., public, private) in both eclipse and netbeans fall within the IQR of respective fields as recorded in Table 4.2.

Distribution of Field Hosting Classes

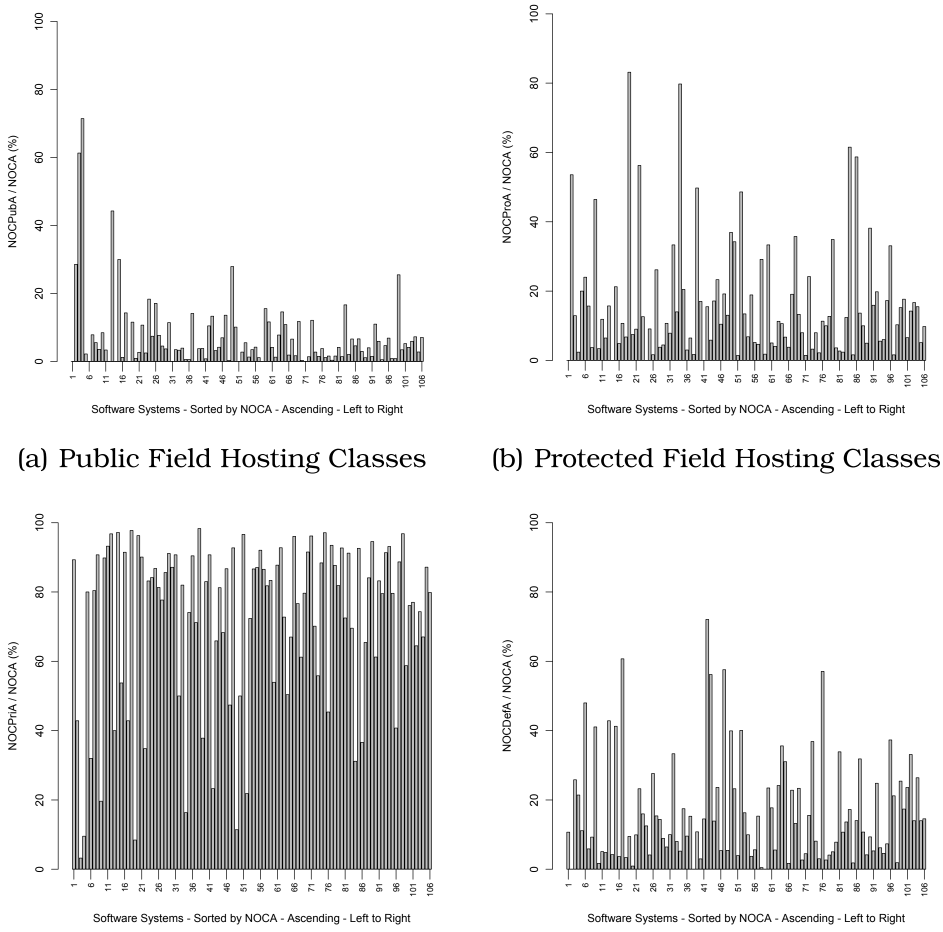
Though the distribution profile of fields provides us with insights into the extent of concentration of fields, that profile does not reveal a system-specific proportion of field hosting classes (NOCA). We do not know yet which software system has a higher proportion of field hosting classes.

Our inspection revealed that the profile of proportions of field hosting classes in the software systems in Qualitas Corpus follows a normal distribution with the mean value 48% (cf. Figure 4.8). Though this suggests that our experimental data set is composed of a diverse collection of software systems that includes fields hosting classes with varying proportions, the profile indicates that the proportion of field hosting classes lies within 40% to 60% in most of the software systems. But what is the decomposition of field hosting classes at the level of different visibility modifiers (i.e.. public, protected, private, and default)?



(a) Histogram of classes with fields. (b) Q-Q plot of classes with fields.

Figure 4.8: Distribution of classes comprising fields in the software systems of the Qualitas Corpus.

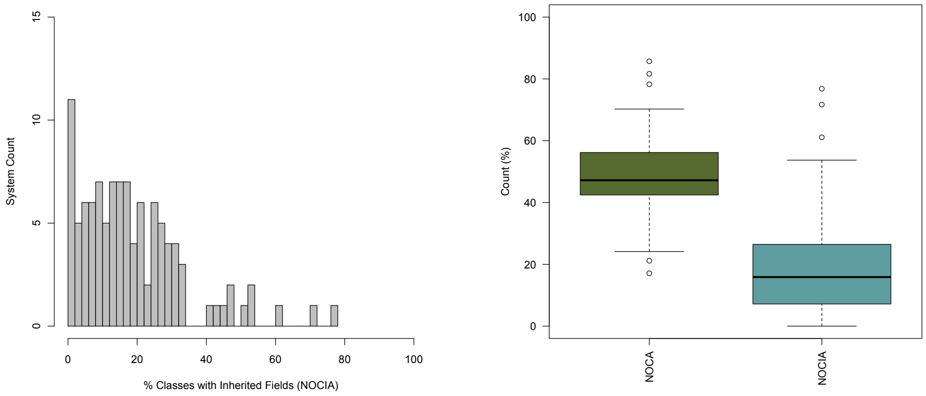


(a) Public Field Hosting Classes (b) Protected Field Hosting Classes

(c) Private Field Hosting Classes (d) Default Field Hosting Classes

Figure 4.9: Classes with different types of fields in the software systems of the Qualitas Corpus.

As depicted in Figure 4.9, we observed that non-private fields hosting classes (i.e., NOCPubA, NOCProA, NOCDefA) are less than 20% in most of the software systems. On the other hand, private fields hosting classes (i.e., NOCPriA) is mostly above 70%. This finding further confirms that when developers define non-private fields, they keep them confined in only a few classes.



(a) Classes with Inherited Fields

(b) Comparison of Classes with Defined and Inherited Fields

Figure 4.10: Classes with Inherited Fields in Qualitas Corpus

In addition to hosting fields, a class may also inherit fields from their ancestor(s). To what extent do classes inherit fields? We found that classes that inherit fields (NOCIA) make up only a small proportion when compared to classes that defined fields. We found that the proportion of such classes ranges between 10% to 30% (cf. Figure 4.10) in most of the software systems. This indicates that developers usually define fields comparatively more than inheriting them from ancestors.

4.3.3 Relation between Volume and Distribution of Fields

The proportion of fields is inversely related to their concentration regardless the visibility modifier (cf. Figure 4.11). It implies that the more fields a software system comprises, the more they are dispersed. Among four types of visibility, the dispersion of non-private fields with

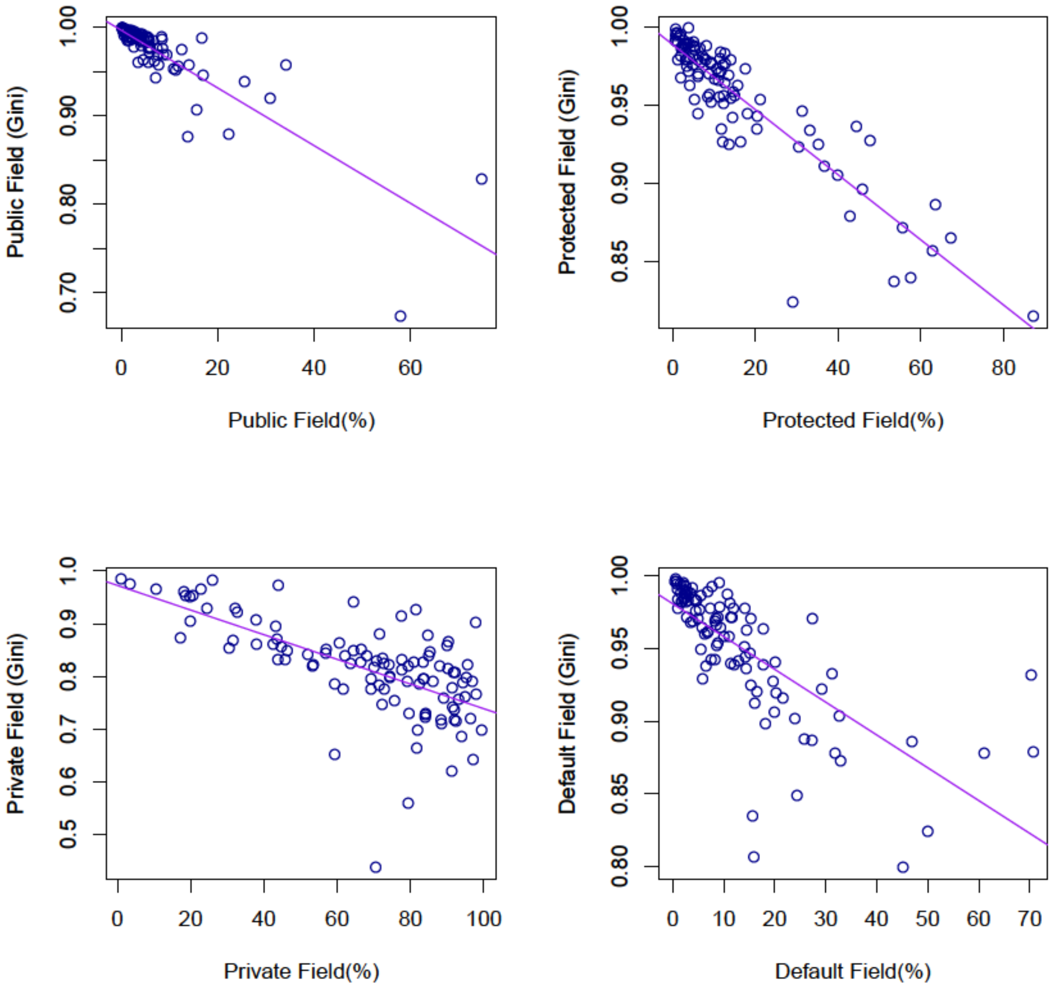


Figure 4.11: Field proportion vs. Gini coefficient.

respect to their proportion varies little within a short interval (i.e., Gini coefficient ranges from 0.7 to 1). This is because they are relatively few in number and also kept highly concentrated in a few number of classes. Even in case of more non-private fields, the concentration does not vary substantially.

The dispersion range of private fields, on the other hand, is a bit wider (i.e., Gini coefficient ranges from 0.4 to 1). This suggests that the private fields are dispersed comparatively more as their proportion increases. But, why do the developers organize the solutions by dispersing the private fields?

The reason of such dispersion of private fields can be attributed to the adaptation with increased complexity as stated by Lehman's second law [129]. According to this law, additional work is necessary for reducing the complexity that arises from software evolution. Though the domains are different, we argue that this complexity can also originate from a variety of sources (like solution design pattern, the way developers organize desired functionality and implement them, etc.). The necessity to cope with complexity, regardless where it originates from, can effect developer preferences and decision making. This coping with complexity can have two aspects: the complexity of the software systems being developed and the cognitive load[3] on developers. Both aspects require proper attention in order to structure a successful development and evolution of a software system [217].

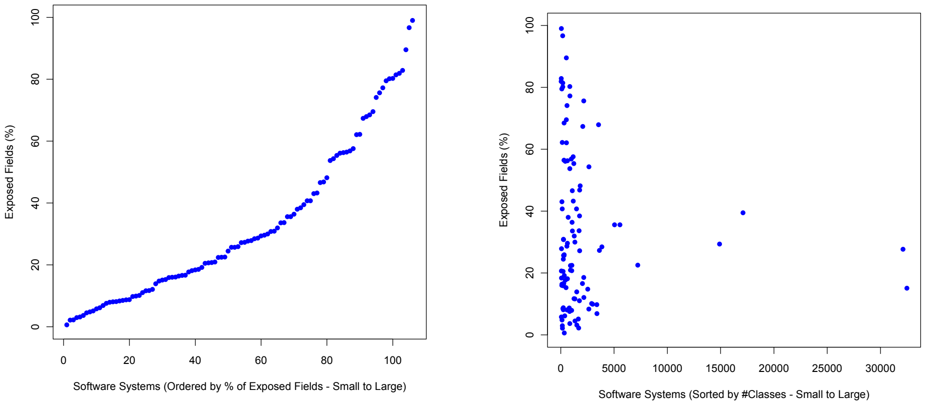
Cognitive load is dependent on element interactivity [203]. When the interactions among different elements are required to be learned, the corresponding cognitive load becomes higher. It can be minimized by learning them successively rather than simultaneously when individual elements do not interact with each other. Based on this, we argue that large classes that comprise a wide variety of functionalities result in high volume of interaction among them. This, in turn, imposes high cognitive load on developers. To reduce the cognitive load of managing such large classes, developers distribute solutions design into different smaller and manageable classes instead of organizing them all together in God-like classes. As a result of such dispersing approach, fields that capture the state of a desired object are distributed across different classes, and this leads to the concentration of fields to be reduced as their proportion rises.

The limitation imposed by the human working memory is another potential reason for such dispersion. Working memory is limited in terms of storage, duration, and processing capacity [27, 167]. According to Cowan [63], working memory can process only 4 ± 1 elements. This may explain the dispersion of fields as their proportion increases.

[3] Total amount of mental activity imposed on human memory.

4.3.4 Field Exposure

An exposed field is a field that has any non-private visibilities (i.e., public, protected, and default visibility) [207]. In this section, we describe (i) the volume of exposed fields, (ii) the degree of exposed field hosting classes, and (iii) the relation between exposed fields hosting classes and exposed fields accessing classes.



(a) Exposed Fields in the Studied Systems

(b) Exposed Fields vs. System Size

Figure 4.12: Exposed Fields Distribution

Volume of Exposed Fields

To gain an insight into the volume of exposed fields used in Java-based software systems, we inspected their proportions in the total instance fields space. Our investigation revealed that the use of exposed fields varies across a wide range (cf. Figure 4.12(a)). But the use of exposed fields in a software system does not depend on its size (cf. Figure 4.12(b)). We observed that software systems sharing almost the same size within the range of 50 to 4,000 classes comprise varying proportions (almost 1 to 99%) of exposed fields. But what are the systems that employ such extensively high (or low) proportions of exposed fields?

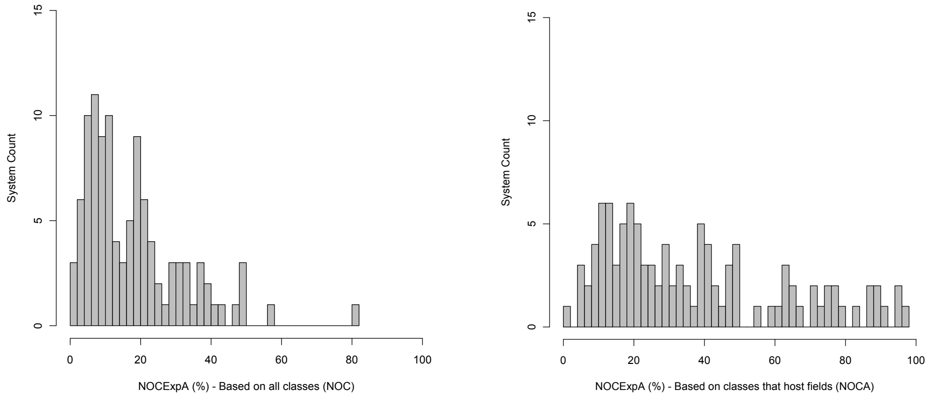
Our investigation revealed that while 10 software systems in the Qualitas Corpus employ less than 5% exposed fields, 10 other software systems employ more than 80% exposed fields. Table 4.3 lists these software systems.

System	Exposed Fields (%)	System	Exposed Fields (%)
checkstyle	0.60	joggplayer	80.13
informa	2.13	jung	80.24
rssowl	2.21	jmoney	81.39
jpf	2.96	jasml	81.87
tapestry	3.17	nekohtml	82.83
jfreechart	3.63	antlr	89.51
myfaces_core	4.48	jgraph	96.64
displaytag	4.80	fitjava	98.99

Table 4.3: Software Systems with less than 5% and more than 80% exposed fields

Degree of Exposed Field Hosting Classes

While the proportions of exposed fields in Java-based software systems ranges from 1 to 99% (approx.), what is the typical profile of classes that host exposed fields? Do the developer design choices confine them in a few classes or do they disperse exposed fields in almost every field hosting classes? To answer these questions, we studied the classes that host at least one exposed field (NOCExpA).



(a) Exposed Field Hosting Classes - (% based on all classes)

(b) Exposed Field Hosting Classes - (% based on classes that host field)

Figure 4.13: Proportion of Exposed Field Containing Classes

We found that a varying range of proportions of classes host exposed fields in almost all the studied software systems. While Figure 4.13 depicts an overview of the extent of classes that comprise the exposed fields, Table 4.4 shows the software systems that employ less than 10% exposed field hosting classes and also more than 80% exposed field hosting classes (NOCExpA).

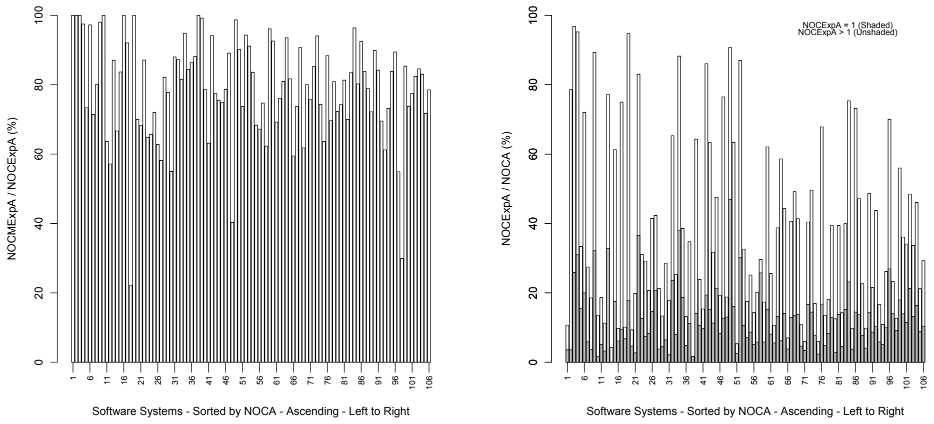
System	NOCExpA/NOCA(%)	System	NOCExpA/NOCA (%)
checkstyle	1.70	jgraphpad	83.04
informa	4.23	pooka	86.05
rssowl	5.34	jung	87.02
tapestry	5.94	colt	88.24
castor	6.05	webmail	89.29
jfreechart	7.01	antlr	90.69
picocontainer	9.35	jgraph	94.74
junit	9.76	jasml	95.24
argouml	9.79	fitjava	96.77
nakedobjects	9.82	-	-

Table 4.4: Software Systems with less than 10% and more than 80% exposed field containing classes (NOCExpA) (Proportion is based on only the classes that host fields).

We observed less than 10% exposed field hosting classes (NOCExpA) in some software systems as a result of only few exposed fields in them. Such classes contain more than 90% private fields. Therefore, the remaining exposed fields are distributed in few number of classes. On the other hand, software systems that employ more than 80% exposed fields hosting classes (NOCExpA) use an extensive amount (more than 70%) of exposed fields. The only exception is jgraphpad that defines 56% exposed fields.

Thus, it appears that a limited proportion of exposed fields are confined in few number of classes. This finding leads to a question: do developers define exposed fields in specially designated classes? If the exposed field hosting classes (NOCExpA) are dominated by exposed fields (NOExpA), then it would indicate that when developers define exposed fields, they tend to define them in classes that usually do not host much private fields.

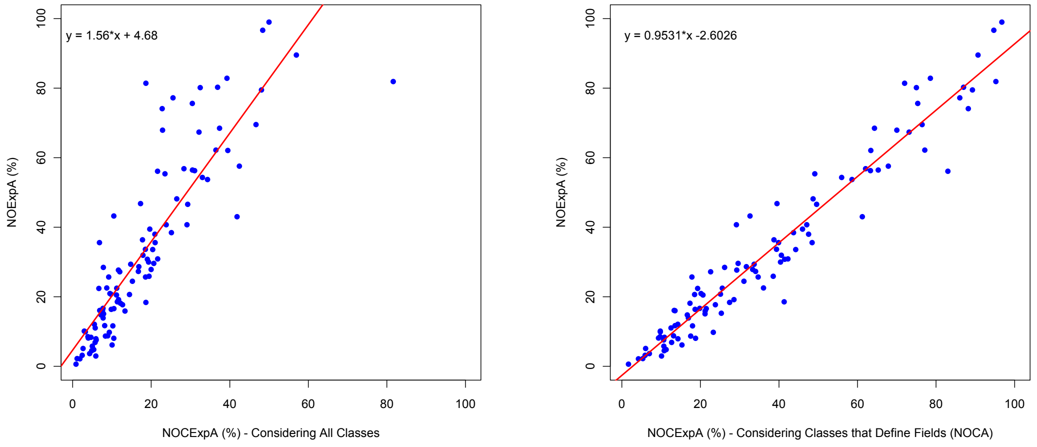
To answer the above question, we studied the extent of exposed fields (NOExpA) in their host classes (NOCExpA). For this purpose, we investigated exposed field hosting classes where more than 50% of the total fields are exposed. The resulting findings (cf. Figure 4.14(a)) indicate that a substantial proportions of NOCExpA are dominated by exposed fields. This means developers tend to expose fields in specially designated classes. In most of the cases, such classes only comprise a single exposed fields (cf. Figure 4.14(b)).



(a) (b)

Figure 4.14: Distribution of classes comprising exposed fields for the QC data (a) Exposed Field Hosting Classes where more than 50% Fields are Exposed, (b) Exposed Field Hosting Classes (where the *shaded* proportion indicates NOCExpA = 1, and the rest implies NOCExpA > 1)

We found a linear relationship between the proportion of exposed fields, and proportion of exposed field hosting classes (cf. Figure 4.15). Though our computation is based on only the classes that host exposed fields, our finding confirms that the more exposed fields are employed in a software systems, the developers design choice (deliberate or accidental) causes more classes to share those exposed fields.



(a) (b)

Figure 4.15: Relation between Exposed Fields and Their Hosting Classes (a) Considering All Classes (b) Considering only the Field Hosting Classes in the Software Systems of Qualitas Corpus

Relation between Exposed Fields Hosting Classes and Exposed Fields Accessing Classes

We also studied exposed fields hosting classes (NOCExpA) and exposed fields accessing classes (NOCWIA) to identify any potential relationship that captures the intention of developers regarding field exposure (i.e., why do they expose fields). The reasons could be twofold: lack of conscious design decision or there exists an intention to use them.

In a recent study, Tempero [207] stipulated that non-private fields may be a result of accidents (or oversights) rather than conscious design decisions as the studied systems do not take advantage of non-private fields. While this finding represents system level scenario, changing the viewpoint could result in a different observation. This is because a particular aspect of software systems may not be visible due to not choosing the right level of granularity [172]. We argue that observations are not always viewpoint agnostic.

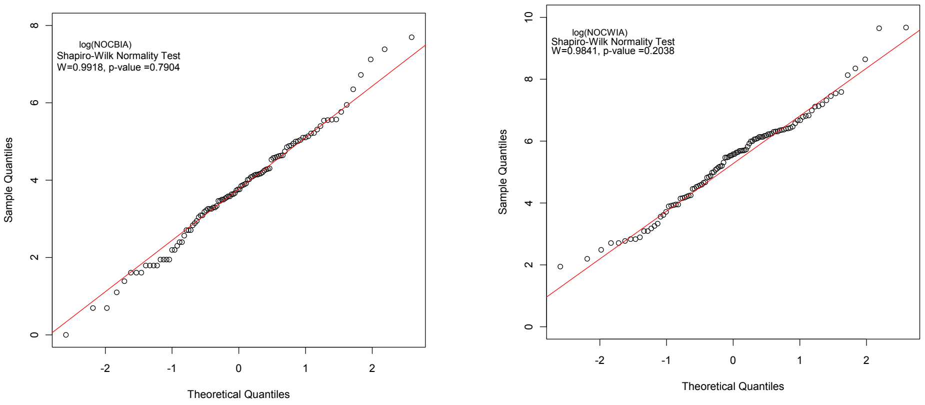
In a recent paper on ecological aspect of empirical software engineering, Posnett et al. [172] emphasize on selecting the right level of granularity. While an observation can be true at one level (e.g., system level), it might exhibit a different scenario at a different level (e.g., entire population level). The characteristics portrayed on an entire population may not be mapped to each constituent systems.

According to Parsons and Wand [164], *things can be combined to form a composite thing. A property of a composite that is not possessed by any of its components is called an "emergent property"*. This is an inherent property of the composite thing, and this property is not localized to any component system. For example, reliability of a software system is considered as an attribute of the overall system, though it can depend on the individual components and the relationship among them.

Emergent properties can be divided into two types: functional and non functional [198]. While the former appears when the system is assembled as a whole by integrating all the constituent components, the later is related to the behavior of the system in its operational environment.

Emergent properties of complex software systems can be revealed at various levels of granularity. For example, Valverde et al. [211] investigated a large collection of object-oriented software systems to uncover any emergent properties at the system level. They discovered *small-world* like networks (i.e., the average distance between any pair of classes is very small) in object oriented software systems.

In order for an emergent property to materialize, a proper level of granularity is required to be applied. We consider the entire population as *system of systems* to uncover any emergent properties related to the field exposure. In particular, we shifted our view point from system level to entire population level to discover any patterns of relationship between the classes comprising exposed fields and the classes accessing them.

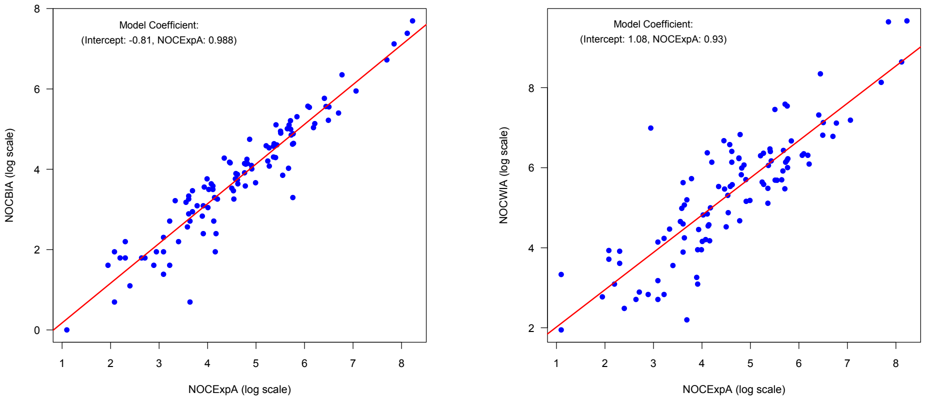


(a)

(b)

Figure 4.16: Both the *Number of classes whose fields are being accessed indirectly from other classes (NOCBIA)*, and the *Number of classes which access fields defined in other classes (NOCWIA)* follow the normal distribution after log transformation.

We found that both the distribution profile of *Number of classes whose fields are accessed indirectly from other classes (NOCBIA)*, and the *Number of classes which access fields defined in other classes(NOCWIA)* follow the normal distribution after log transformation (cf. Figure 4.16). Though the existence of any one to one mapping between these two distributions is yet unknown, we argue that the two measures (i.e., NOCBIA and NOCWIA) are related as they follow the same distribution pattern.



(a)

(b)

Figure 4.17: (a) Relation between *Number of classes containing at least one exposed field (NOCExpA)* and *Number of classes whose fields are being accessed indirectly from other classes (NOCBIA)*. (b) Relation between *Number of classes containing at least one exposed field (NOCExpA)* and *Number of classes which access fields defined in other classes (NOCWIA)*.

In addition, the above two measures (i.e., NOCBIA and NOCWIA) are exponentially related to the *Number of classes containing at least one exposed field (NOCExpA)* (cf. Figure 4.17). This is an emergent property observed at the entire population level, not necessarily visible at the system level.

4.4 Summary

In this chapter, we studied how developers use fields in Java-based software systems. In particular, we investigated developer tendencies in adhering to the data hiding principle [162] and availablle advices (e.g., [12, 183]) regarding the use of fields. Moreover, we studied associated design choices including organization of data storage (as represented by fields) across different classes, and data exposure profiles (as indicated by fields with non-private visibility modifiers).

We observed the following:

- Developers tend to follow advices regarding limiting the use of non-private fields (e.g., *all data should be hidden within its class* [183], *don't expose state if you don't have to* [12]). This observation is evident in two levels of granularities: the entire collection of software systems in Qualitas Corpus, and also across constituent domains (which suggests that developers of any particular domain also adhere to advices).
- Developers do not define fields in all classes. We found fields in 40 to 60% of the classes in most of the software systems of Qualitas Corpus. Apart from this, about 10% to 30% of the classes in most of the software systems comprise inherited fields.
- Developer design choices are centralized in terms of data storage (as represented by fields) in typical Java-based software systems. When developers expose fields, they confine them in a few classes. The Gini coefficient of such fields is above 0.9 in more than 80% software systems in the Qualitas Corpus.
- The dispersion of fields is related to their proportion. This indicates that developers do not tend to create God-like classes (in terms of fields), rather they usually work with smaller and manageable classes. One underlying reason may be an increased cognitive load associated with managing classes with high field count.
- Exposed field hosting classes are mostly exposed field dominated. When developer define exposed fields in a class, they tend to limit the private fields in that class.
- We established a relation between exposed fields and their usage. The distribution profile of both the *number of classes whose fields are accessed*, and *who access those fields* follow a normal distribution (after log transformation) at entire population level (i.e., not at the level of each software system, rather their collection as a whole). Given the absence of a one to one mapping between these two distributions, it is still evident that both follow the same pattern. We found both the *classes whose fields are accessed*, and

classes who accesses fields of other classes indirectly is an exponential function of the *classes with exposed fields*. This reinforces our observation that the exposure of fields is somewhat deliberate (as exposed fields are being accessed to some extent) and somewhat accidental.

Chapter 5

Property Analysis

In this chapter, we focus on investigating developer behavior in using the property mechanism in Java. To begin, we set the stage for studying the property mechanism and identify a set of specific research questions. We then describe the analysis method in section 5.2. This section presents a set of definitions that capture different usage patterns of properties practiced by developers, a corresponding set of software metrics, and the description of our metrics analysis approach. We present the results of our investigation as observations in section 5.3. Finally, we summarize the key findings of this chapter in section 5.4.

5.1 Introduction

In the previous chapter, we studied the use of fields along with different visibility modifiers that allow us to fabricate controlled access to an object's internal state, and thus assist us to ensure data encapsulation. However, there is another dimension of data encapsulation that we also need to consider – *getters* and *setters* – designated member functions to access and manipulate the state of objects. Getters and setters (aka, accessor and mutator, respectively) are collectively called *properties* and have been popularized in the 1990s, when they were introduced as a new linguistic element to Borland's *Delphi* object model [48]. Though possible, the purpose of getter and setter methods is not to circumvent the visibility modifiers, in particular *private*, but to provide a means to effectively convey the intent of code while at the same time giving developers the tools to retain the level of protection generally associated with information hiding.

Though many object-oriented programming languages (e.g, C#, Object Pascal) provide built-in support for the property mechanism, the Java programming language lacks it [26]. As a result, Java developers are required to fabricate the property model. While accomplishing this task, they are expected to adhere to a *coding convention* [10] or *architectural style* [191] that comprises well defined guiding principles for this purpose (e.g., prefixing a field name with *get* and *set* while defining getter and setter methods).

While the conventions prescribe how one should define getter and setter methods, there is a considerable amount of advices available that suggest whether one should use them or not. For example, Reil [183] encourages to change the state of an object through its public interface (i.e., getter and setter methods). On the other hand, many emphasize to avoid them whenever possible. For example, Holub [105] considers them as *evil* as they violate the encapsulation principle. Similar opinion is also expressed by Eby [69] and Jorgensen [114].[1]

One basis of such an opinion is that a well-encapsulated class should not expose its internals through getter and setter methods. Such exposure is often considered as an indication of lack of good object-oriented design [114]. Moreover, such exposure is blamed for not only violating encapsulation, but also incurring difficulty in software maintenance [105, 106]. In particular, when getter and setter methods are used only to access or mutate a field, the field becomes exposed like being a public one. There is not *much* difference between such use of getter and setter methods and a public field - both violate the encapsulation principle. Yet, the use of getter and setter methods may offer some advantages when compared to public fields. For example, changes in an object's

[1] These opinions are published on several websites. Though websites may not be the most reliable sources of advice, they represent at least the viewpoint of well-experienced developers in this case.

internal implementation may result in the field to change, and thus allow the client code that rely on associated getter and setter methods to remain unaffected.

But, in a statically typed language like Java, there is still room for problems: a change in the *type* of a field may cause the client code to break. For example, altering the type of an instance field *Color* from *Int* to *String* demands change in associated getter and setter methods (if any), and thus enforces changes in the corresponding client code (e.g., dependent classes). A dynamically typed language, however, does not suffer from such a problem. For example, Python [214] offers properties to control access to instance variables. In Python, the interface for accessing a variable that is public and a variable whose access is controlled by properties is the same. As a result, a client's code may remain unaffected when an object's internal implementation changes.

However, getter and setter methods may not simply be used to access and modify associated fields. As Java allows flexibility to define getter and setter methods by supporting them through conventions, in addition to merely getting and setting a field, developers may choose to incorporate more functionality (e.g., implementation of additional domain logic, lazy initialization, logging) in getter and setter methods. Though possible in Java (unlike C#), such usage of getter and setter methods somewhat mismatches with their primary purpose.

Thus, programming language-specific idioms may govern the use of properties. Coding conventions like Java's naming pattern for getter and setter methods can give rise to a systematic bias towards the way developers formulate solution design [140]. They may either ease or make difficult the way developers express intent in software. In addition, the available advices regarding the use of properties can affect developer preferences, and thus may influence them towards particular design choices. But how do the developers *actually* use them? In this chapter, our focus is on understanding developer practices in using the property mechanism in Java.

In particular, we address the following research questions:

- **RQ6**: What is the typical distribution profile of properties in Java-based software systems? Do developers adhere to the Java-specific coding conventions and associated guidelines while formulating solutions using getter and setter methods?
- **RQ7**: Is there any impact of the underlying problem domain on the use of getter and setter methods?
- **RQ8**: Do developers define getter and setter methods when they define private fields?
- **RQ9**: What is the distribution of ratio of getter and setter methods?

5.2 Analysis Method

To facilitate our analysis, we classify getter and setter methods into 12 different types that capture their different variants (cf. Figure 5.1). We then define a set of software metrics that capture these definitions and analyze the defined metrics with different measures (e.g., the Gini coefficient). In this section, we present (i) definitions of variants of getter and setter methods, (ii) definitions of corresponding metrics, and (iii) the data analysis approach used.

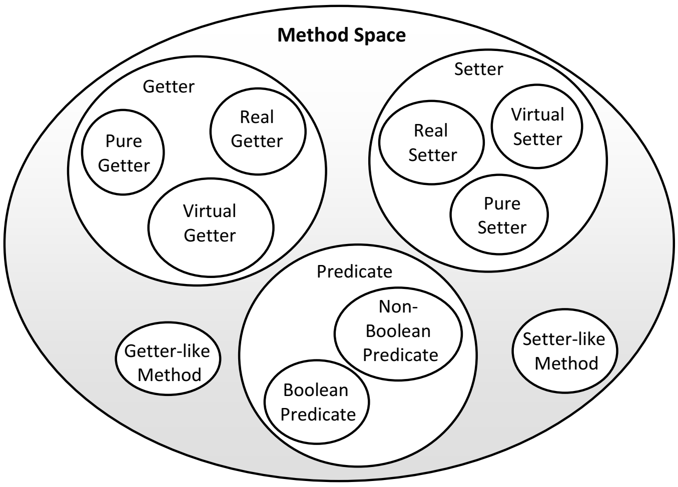


Figure 5.1: Method Space with different Getter and Setter methods (Entities are not being mapped to scale)

5.2.1 Definitions

Getters

A unique method whose name starts with the prefix “get". The Java code of an example of getter methods is given below.

```
public String getName()
{
   //more code(optional) - pre-condition
   return fName;
}
```

The bytecode sequence of a getter method can have a variety of patterns, depending on the purpose the getter is used for. For example, while a getter method can be used only to access a particular field, it can also apply additional computations to the retrieved field.

Setters

A unique method whose name starts with the prefix “set". The Java code of an example of setter methods is given below.

```
public void setName (String aName)
{
   //more code(optional) - pre-condition
   fName = aName;
   //more code(optional) - post-condition
}
```

Like getter methods, the bytecode sequence of a setter method can assume different patterns, depending on its purpose. For example, while a setter method can be used only to modify a field, it can also be used to implement different functionalities before and after[2] setting the associated field.

[2] A *bound* setter, for example, notifies listeners when its value changes [21].

Pure Getters

A getter method that is used to access the target field only. We call it *pure* as it does not offer any additional functionality rather than providing access to the target field. It is the simplest form of all the getter methods. An example of pure getter methods is given below.

```
public String getName()
{
   return fName;
}
```

The bytecode representation of pure getters is presented below.

```
   0: aload_0
   1: getfield   #2; //Field fName:Ljava/lang/String;
   4: [ i | l | f | d | a] return
```

Pure Setters

A setter method that is used to modify the target field only. We call it *pure* as it does not offer any additional functionality rather than providing modifying the target field. It is the simplest form of all the setter methods. An example of such pure setter methods is given below.

```
public void setName (String aName)
{
   fName = aName;
}
```

The bytecode representation of pure setters is presented below.

```
   0: aload_0
   1: aload_1
   2: putfield #2; //Field fName:Ljava/lang/String;
   5: return
```

Real Getters

A getter method that is used to access the target field of its host class. In addition to providing access to the desired field, this type of getter method can offer additional computations, if necessary. An example of real getter methods is given below.

```
public String getName()
{
   //more code(optional) - pre-condition
   return fName;
}
```

Both the getter - *getName()* and the field - *fName* are defined in the same class (i.e., the getter accesses the host class's field).

Real Setters

A setter method that is used to modify the target field of its host class. Like a real getter, the real setter method can also employ some additional processing before modifying the field. An example of real setter methods is given below.

```
public void setName (String aName)
{
   //more code(optional) - pre-condition
   fName = aName;
   //more code(optional) - post-condition
}
```

Both the setter - *setName()* and the field - *fName* are defined in the same class (i.e., the setter modifies the host class's field).

Virtual Getters

A getter method that is used to access the target field that is defined outside of its host class. In addition to providing access to the desired field, this type of getter method can offer additional computations, if necessary. An example of virtual getter methods is given below.

```
public String getName()
{
   // code retrieving information associated with an actual
   // (not defined in the host class) or emulated field.
}
```

Virtual Setters

A setter method that is used to modify the target field that is defined outside of its host class. Like a virtual getter, the virtual setter method can also employ some additional processing before modifying the field. An example of virtual setter methods is given below.

```
public void setName (String aName)
{
   // code updating information associated with an actual
   // (not defined in the host class) or emulated field.
}
```

Predicates

Predicate can be viewed as a special purpose getter function that returns a Boolean value. In other words, rather than returning the associated field value, a predicate tests for a specific object state associated with one or more fields. To assess the relation of this type of state access with respect to other getter variants, we included this type in our study, even though predicates are rather function than properties. An example of Boolean predicates is given below.

```
public boolean isEven()
{
   return (fNumber % 2) == 0;
}
```

The predicate - *isEven()* checks whether the field *fNumber* is even or not, and return a Boolean result.

Non-Boolean Predicates

A predicate that returns a non-boolean value is called a non-boolean predicate. An example of such predicates is given below.

```
public int isEven()
{
   if (fNumber % 2) == 0)
      return 1;
   else
      return 2;
}
```

The predicate - *isEven()* checks whether the field *fNumber* is even or not, and returns a non-boolean result.

Getter-Like Methods

Often a method is providing the functionality of a getter method, though its name does not reflect it. That is, the method name is not prefixed with *get*, but it does the same job as a getter does. We call them *getter-like* methods. We identified this type of method by comparing their compiled image with that of a pure getter. We restricted our scope to getter-like methods that have three instructions only, though there might be even more of them. An example of getter-like methods is given below.

```
public String myMethod()
{
  return fName;
}
```

The byte code representation of getter-like methods is presented below.

```
   0: aload_0
   1: getfield  #2; //Field fName:Ljava/lang/String;
   4: [ i | l | f | d | a] return
```

Setter-Like Methods

Like the getter-like methods, there are methods that do the same job as a setter, but their names do not reveal it (i.e., the name of the method does not start with *set*). The compiled image of such method is same as that of a pure setter. Though there might be even more of them, we counted setter-like methods that contain four instruction only. An example of setter-like methods is given below.

```
public void yourMethod (String aName)
{
  fName = aName;
}
```

The bytecode representation of setter-like methods is presented below.

```
   0: aload_0
   1: aload_1
   2: putfield #2; //Field fName:Ljava/lang/String;
   5: return
```

5.2.2 Definitions of Metrics

To facilitate analysis of the property mechanism, we mapped the above definitions to a set of software metrics. As getter and setter methods are one type of special purpose methods, we included a set of methods measures in our analysis. One resulting benefit of including methods is that we can conduct comparative analysis of methods, and getter and setter methods to reason about their distribution patterns in Java-based software systems. Table 5.1 presents our rationale.

Table 5.1: Rationale for the Selected Key Measures

Name	Rationale	Description
Getters/Setters	Data Access/Modify	Stored data retrieval/modififcation
Methods	Data Consumption/Production	Stored data consumed/New data produced
	Decomposition	Breadth of functional decomposition

The key focus of the method and property measures is depicted in Figure 5.2. The method measures are concerned with revealing functionality decomposition patterns (in terms of total number of methods defined in a class) with 4 different visibilities modifiers (i.e., public, protected, private, and default).

The property measures focus on capturing different variants of getter and setter methods, mainly covered under four different facets described below:

- *Number of Getters*, the total number of getter methods in a class, and its semantic variants *Number of Real Getters*, *Number of Virtual Getters*, and *Number of Pure Getters* to record specific field access idioms,

- *Number of Setters*, the total number of setter methods in a class, and its semantic variants *Number of Real Setters*, *Number of Virtual Setters*, and *Number of Pure Setters* to count specific field update idioms,

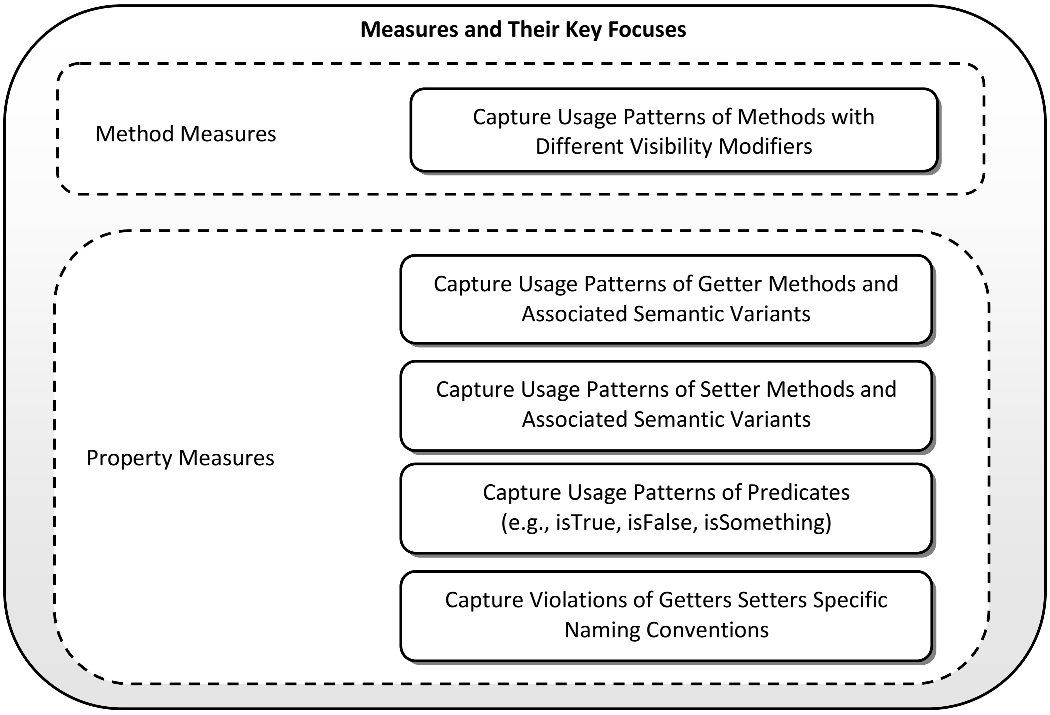

Measures and Their Key Focuses
Method Measures
Capture Usage Patterns of Methods with Different Visibility Modifiers
Property Measures
Capture Usage Patterns of Getter Methods and Associated Semantic Variants
Capture Usage Patterns of Setter Methods and Associated Semantic Variants
Capture Usage Patterns of Predicates (e.g., isTrue, isFalse, isSomething)
Capture Violations of Getters Setters Specific Naming Conventions

Figure 5.2: Measures and Their Key Focuses

- *Number of Boolean Predicates* and *Number of Non-Boolean Predicates*, the total number of predicates in a class, and
- *Number of Pure Getter-like Methods* and *Number of Pure Setter-like Methods* methods that behave like “get"- and “set"-methods but do not follow the prescribed naming convention.

The exact definitions of the corresponding software metrics are shown in Table 5.2. In this table, we defined a set of 15 different software metrics and their relations. The metrics set comprises of 5 *method measures* and 12 *property measures.*

5.2.3 Data Analysis Approach

Our analysis of the above software metrics data is descriptive in nature. We use the measures (e.g., the Gini coefficient, bounded region of functionality distribution as represented by a series of Lorenz curves) presented in the previous chapter. In addition, we conduct a frequency

Table 5.2: The collected method and property measures.

Method Measures		
Name	**Purpose**	**Relation**
Number of Methods (NOM)	Counts all defined member methods in a class	-
Number of Public Methods (NOPubM)	Counts all public methods in a class	NOPubM $\subseteq$ NOM
Number of Protected Methods (NOProM)	Counts all protected methods in a class	NOProM $\subseteq$ NOM
Number of Private Methods (NOPriA)	Counts all private methods in a class	NOPriM $\subseteq$ NOM
Number of Package Methods (NODefM)	Counts all methods with default or package visibility in a class	NODefM $\subseteq$ NOM

Property Measures		
Name	**Purpose**	**Relation**
Number of Getters (NOG)	Counts all member functions in a class, whose name starts with the prefix “get"	NOG $\subseteq$ NOM
Number of Setters (NOS)	Counts all member functions in a class, whose name starts with the prefix “set"	NOS $\subseteq$ NOM
Number of Real Getters (NORG)	Counts all getter methods that access a field defined in the host class	NORG $\subseteq$ NOG
Number of Virtual Getters (NOVG)	Counts all getter methods that access a field defined outside in the host class	NOVG $\subseteq$ NOG
Number of Pure Getters (NOPG)	Counts all getter methods with 3-instruction sequence `aload_0`, `getfield`, and `[i|l|f|d|a]return`	NOPG $\subseteq$ NOG
Number of Real Setters (NORS)	Counts all setter methods that alter a field defined in the host class	NORS $\subseteq$ NOS
Number of Virtual Setters (NOVS)	Counts all setter methods that alter a field defined outside in the host class	NOVG $\subseteq$ NOS
Number of Pure Setters (NOPS)	Counts all setter methods with 4-instruction sequence `aload_0`, `aload_1`, `putfield`, and `return`	NOPS $\subseteq$ NOS
Number of Boolean Predicates (NOBP)	Counts all methods that start with prefix “is” and return a boolean value	NOBP $\subseteq$ NOM
Number of Non-Boolean Predicates (NONBP)	Counts all methods that start with prefix “is” and return a non-boolean value	NONBP $\subseteq$ NOM
Number of Pure Getter-like Methods (NOGLM)	Counts all methods that act like pure getters (i.e., have 3 instructions), but whose method names is not prefixed with “get"	NOGLM $\subseteq$ NOM
Number of Pure Setter-like Methods (NOSLM)	Counts all methods that act like pure setters (i.e., have 4 instructions), but whose method names is not prefixed with “set"	NOGLM $\subseteq$ NOM

histogram-based analysis. Moreover, we use Spearman’s rank correlation coefficient to check for any correlation between a pair of software metrics (e.g., private fields, and getter methods).

5.3 Observations

In this section, we present the outcomes of our empirical investigation of the usage patterns the property mechanism in the Java programming language. This section is divided into (i) overall metrics distribution profiles, (ii) proportion of getter and setter methods, (iii) relation between private fields and getter and setter methods, and (iii) ratio of getter and setter methods.

5.3.1 Overall Metrics Distribution Profile

How do developers use the variants of the property mechanism? Do developer practices cause certain classes to encapsulate most of the functionalities? To answer these questions, we studied the distribution patterns of various software metrics and computed their Gini coefficients.

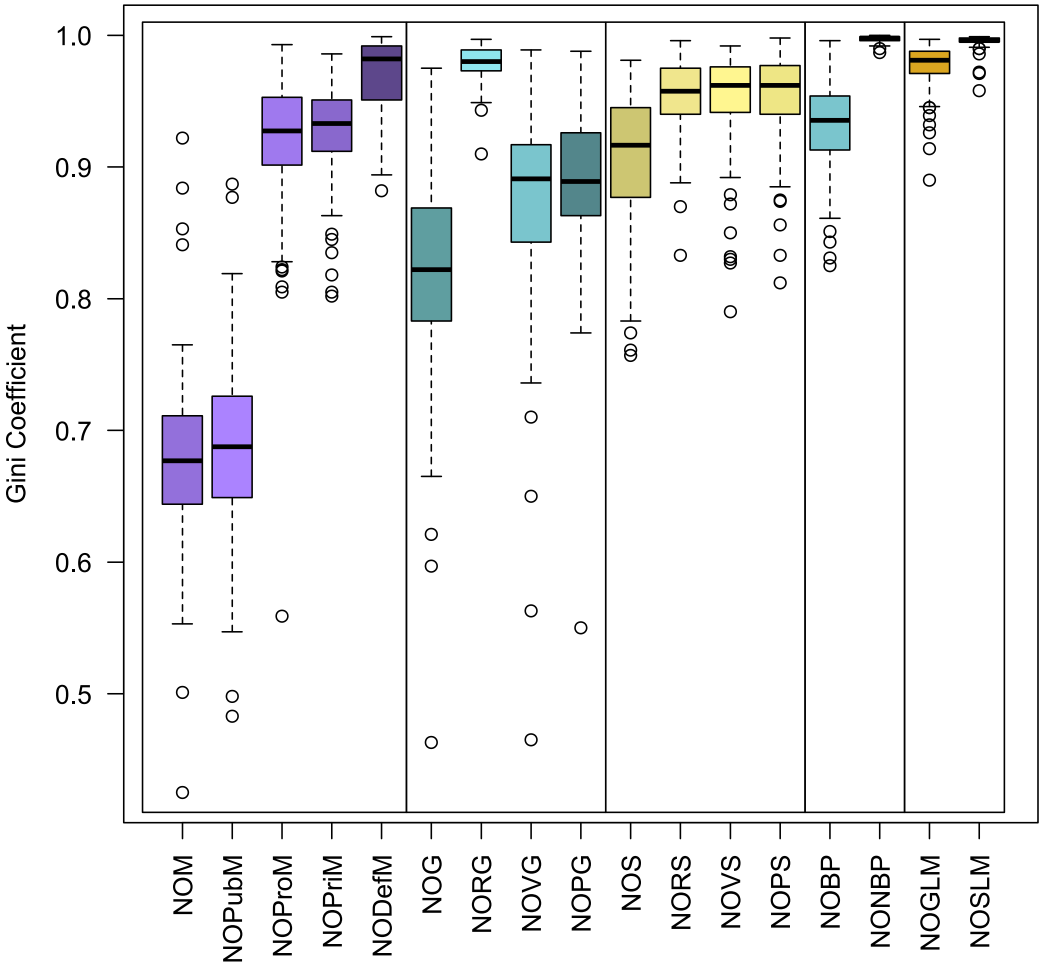


Figure 5.3: Boxplot of all the counted measures.

The range of Gini coefficients of each measure is depicted, in terms of box plots, in Figure 5.3. The box plots indicate that the Gini coefficients are very high, and bounded within a narrow range. Such narrow boundaries of the Gini coefficients indicate a *certain consistency* among the developers in employing the property mechanism available in Java. This finding also implies that the associated distribution preferences may not be a factor of the underlying problem domain (as most of the software systems in Qualitas Corpus, belonging to a variety of domains, show similar patterns of distribution).

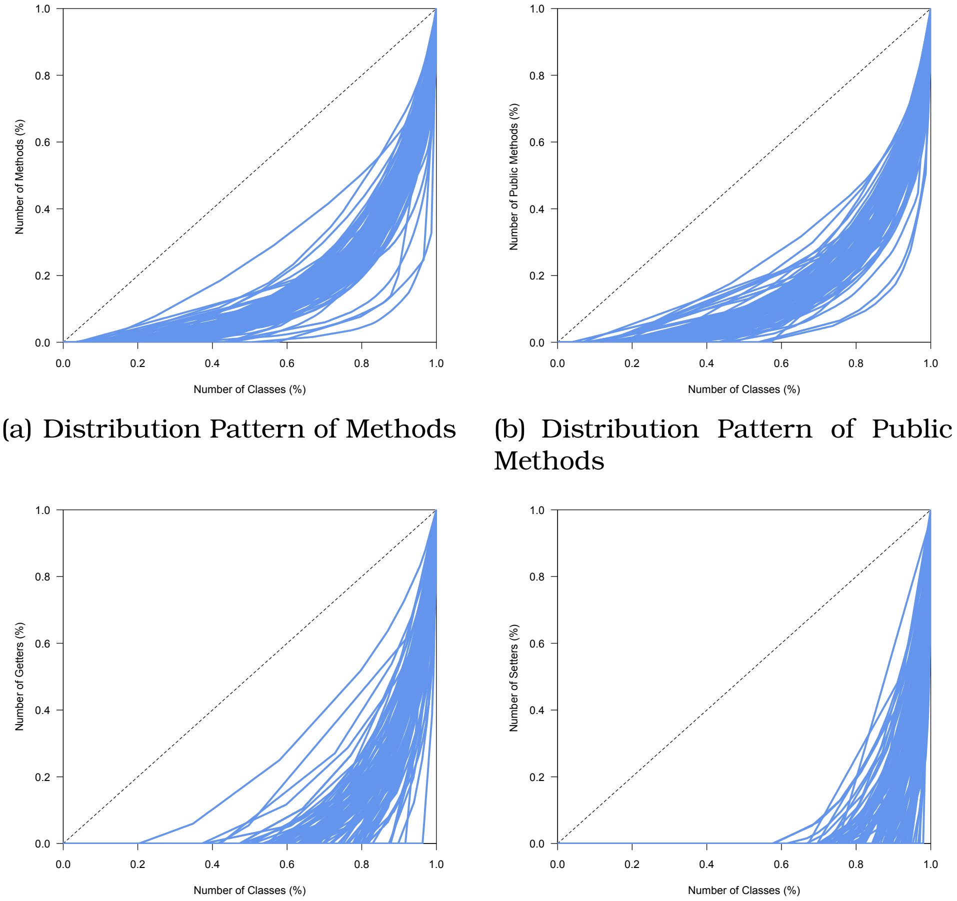
Measure	**IQR - Interval**	**IQR - Width**
Number of Methods	$[0.644, 0.712]$	0.068
Number of Getters	$[0.783, 0.868]$	0.085
Number of Setters	$[0.877, 0.945]$	0.068

Table 5.3: Width of bounded regions of key measures.

To identify the typical region of property metrics distribution, we plotted their corresponding Lorenz curves in a similar fashion described in previous chapter (cf. Section 4.2). Figure 5.4 depicts the results. The intervals of the IQRs of the key measures, presented in Table 5.3, suggest that developers are consistent in organizing their solutions when using getter and setter methods. For all intents and purposes, the IQR width is very small. Developers appear to control their solution designs within very narrow margins. We observed, however, higher concentration of getter and setter methods. Also, the variability is much greater than those of fields (cf. Table 4.2).

Methods

The distribution of methods is the least concentrated one among all the metrics (cf. Figure 5.3). Interestingly, public methods follow almost the same distribution pattern (cf. Figure 5.4(a) and Figure 5.4(b)). On the other hand, methods with non-public visibility modifiers (i.e., protected, private, and default) are highly concentrated (the Gini coefficients are greater than 0.9 in most of the cases). This indicates that developers define them in very few classes.



(a) Distribution Pattern of Methods

(b) Distribution Pattern of Public Methods

(c) Distribution Pattern of Getters

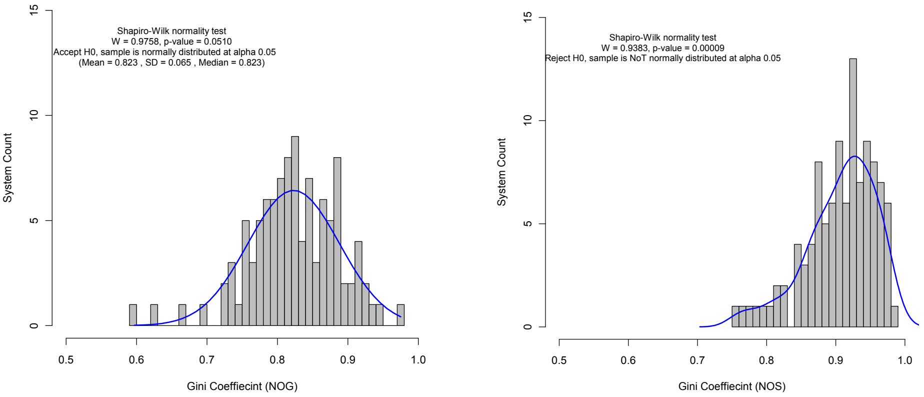
(d) Distribution Pattern of Setters

Figure 5.4: Distribution patterns of selected key metrics - It suggests that the distributions lie within narrow and bounded regions.

The Gini coefficients for methods range from 0.42 (i.e., jparse) to 0.92 (i.e., colt). The interquartile range (IQR) is 0.64 to 0.71 (with a median value of 0.67). The mean value is 0.68 which is slightly higher than the one observed by Vasa et al. [217]. The underlying reasons can be attributed to the associated data sets in the two studies. While Vasa et al. [217] investigated various releases of only 40 software systems from an evolution perspective, our data set (i.e., Qualitas Corpus) comprises 106 different software systems from 12 major domains (e.g., database, middleware). It may be the *diversity* in the data sets that influenced the result. As several releases of a particular software system are less likely to induce significant diversity, we consider the data set used in the study of Vasa et al. [217] is less diverse than the Qualitas Corpus.

Getter and Setter Methods

The Gini coefficients for getter methods range from 0.46 (i.e., jparse) to 0.98 (i.e., colt). The interquartile range (IQR) is 0.783 to 0.868, with a median value of 0.82. On the other hand, the Gini coefficient of setter methods range from 0.75 (i.e., trove) to 0.98 (i.e., jruby). The interquartile range (IQR) is 0.877 to 0.945, with a median value of 0.92.



(a) Getter Methods Gini Distribution

(b) Setter Methods Gini Distribution

Figure 5.5: Gini coefficient distribution of getter and setter methods.

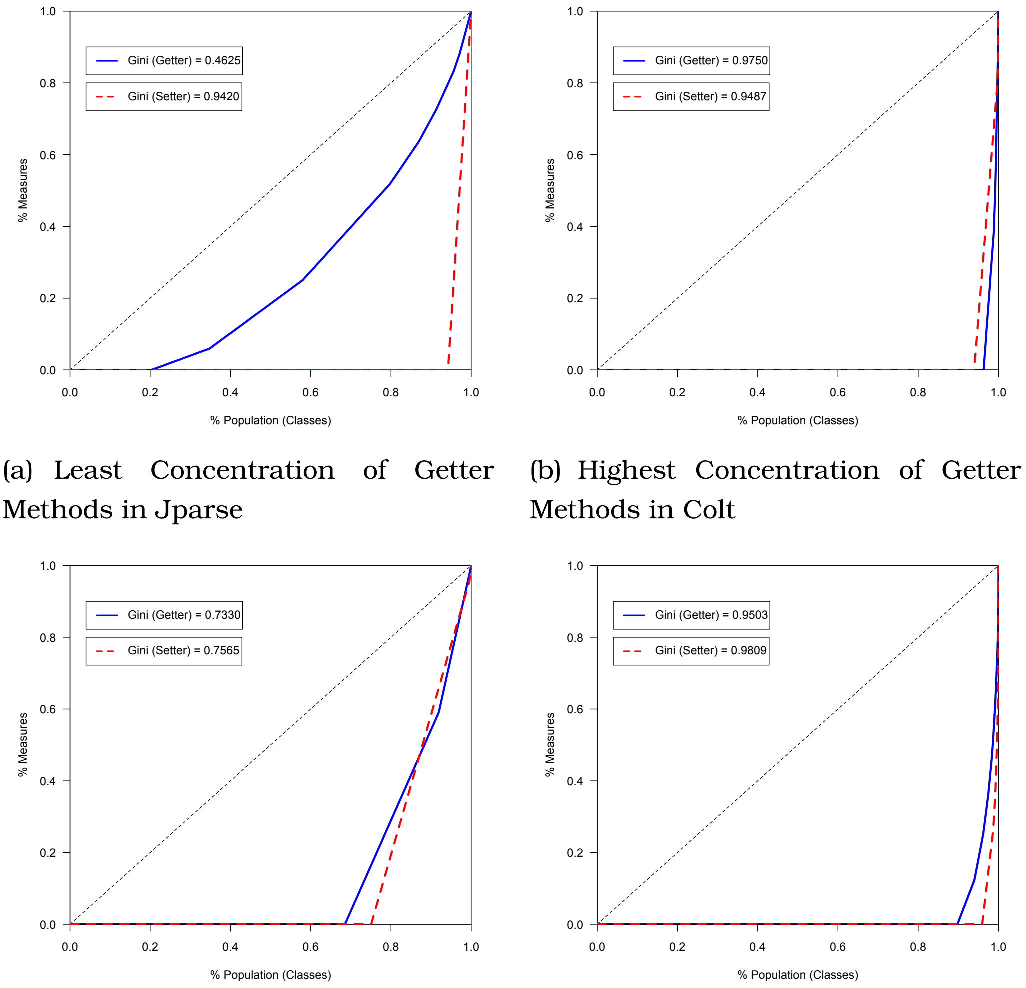
While we observed that the Gini coefficients for getter methods are normally distributed (cf. Figure 5.5(a)), the Gini coefficients for setter methods show a skewed distribution (cf. Figure 5.5(b)). Normal distributions arise when the underlying data is sufficiently large and the population is adequately diverse. A skewed distribution, on the other hand, emerges when a few factors contribute multiplicatively [133]. The getter method Gini coefficients are normally distributed, suggesting an independence from domain and solution design. The skewed nature of the setter method Gini coefficients, however, stipulates the opposite.

Semantic Variants

Semantic variants of getter and setter methods refer to the variety of patterns (e.g., real, virtual) that developers employ while formulating solution design using property mechanism in Java. We found evidence of their presence with varying degree of concentration across the software systems in the Qualitas Corpus. We observed a higher concentration of real getter and real setter compared to virtual getter and virtual setters. The more widely distributed virtual getter and virtual setters suggests that the developers make intentional use of the exposed fields from other classes.

The underlying reasons of using exposed fields through virtual getter setter methods can be many. For example, functionality implemented in one class may require data hosted in other classes. Another possible reason could be underlying design choices of developers. They may decompose a solution design into several aspects (e.g., data storage module, functionality implementation module). For example, certain classes can be used only to store data in a software system, and other client classes can rely on such data holders for implementing functionality. In such a case, the presence of virtual getter and virtual setter methods can significantly exceeds the amount of real getters and real setters in a software system. A similar scenario is observed in the visual component library (VCL) of Delphi [48]. In Delphi, the service superclasses provide the basic functionality that the component subclasses require. For this purpose, the super-class fields are exposed through public get- and set-methods in component subclasses.

An aspect of more concern is that almost all systems also contain methods with get- and set-semantics (i.e., getter-like and setter-like), but are not designated as such. However, the concentration of such methods is very high (above 0.92). This suggests that these semantic variants offer little value to developers and are consequently rare in practice (i.e., resulting in a very high concentration in a few classes). The high Gini coefficient is indicative of what could be viewed as “accidental use”.



(a) Least Concentration of Getter Methods in Jparse

(b) Highest Concentration of Getter Methods in Colt

(c) Least Concentration of Setter Methods in Trove

(d) Highest Concentration of Setter Methods in Jruby

Figure 5.6: The least and the highest concentrations of getter and setter methods in the Qualitas Corpus.

Exceptional Cases

There are some software systems in the Qualitas Corpus that appear as exceptions. We found them in almost all of the measures (e.g., methods, getters, setters). While jparse exhibits the least concentration (Gini coefficient = 0.42), javacc (Gini coefficient = 0.92) has the most concentrated distribution profile for methods. These software systems are rather small in size (less than 150 classes), and therefore the method distribution found in them can be considered as exception and may not necessarily be representative of developers practices.

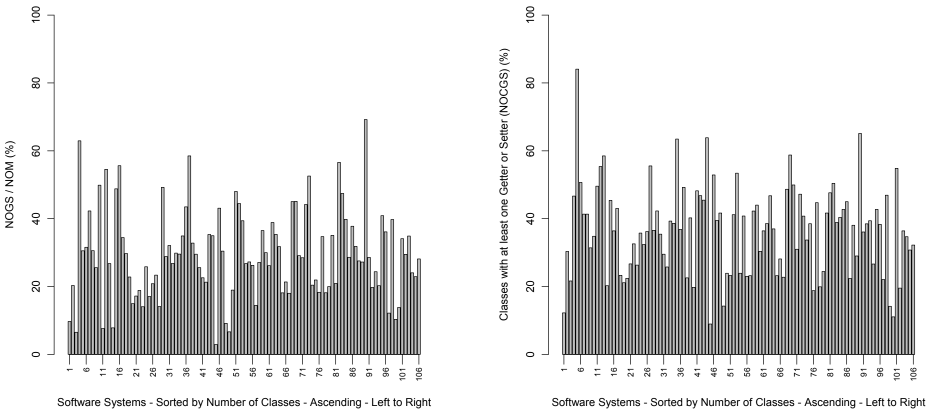
In case of distribution of getter methods, the software system with the least concentration of getter methods is again jparse. It defines almost 30% of its total methods as getter methods which are dispersed in 80% of its classes (cf. Figure 5.6(a)). On the other hand, colt has the highest concentration of getter methods. This software system use only a few (i.e., < 50) getter methods. In colt, 85% classes do not comprise any getter methods at all (cf. Figure 5.6(b)).

The software system with the least concentration of setter methods is trove. It defines less than 200 setter methods that are distributed in only 25% classes (cf. Figure 5.6(c)). The highest concentration of setter methods is observed in jruby, even though it contains 5055 classes. Only 3% of classes in jruby define setter methods (cf. Figure 5.6(d)).

However, such exceptional Gini coefficients (that lie outside the IQR interval) do not necessarily signal problems, but rather specific, possibly domain-dependent, design decisions. Whether or not to trigger alarms (that indicate the necessity of potential actions) in these cases depend on project-specific settings. We cannot predict the initial value of the Gini coefficients for software metrics data. But once a Gini coefficient has been recorded for a particular measure, it moves little, typically less than 0.1 over the lifetime of a system, but often within the bounds of the associated regions [217].

5.3.2 Proportion of Getter and Setter Methods

Our investigation revealed that developers define getter and setter methods with a varying degree of proportions that range from 2.95% (colt) to 69% (compiere). Figure 5.7(a) demonstrates the variability in proportion of getter and setter methods in software systems of the Qualitas Corpus. This implies that the use of getter and setter methods does not depend on the system size size (in terms of number of classes), and they are hosted in varying proportions of classes (cf. Figure 5.7(b)).



(a) Proportion of Getters-Setter Methods

(b) Proportion of Getters-Setter Hosting Classes

Figure 5.7: Getter and Setter Methods Distribution in Qualitas Corpus.

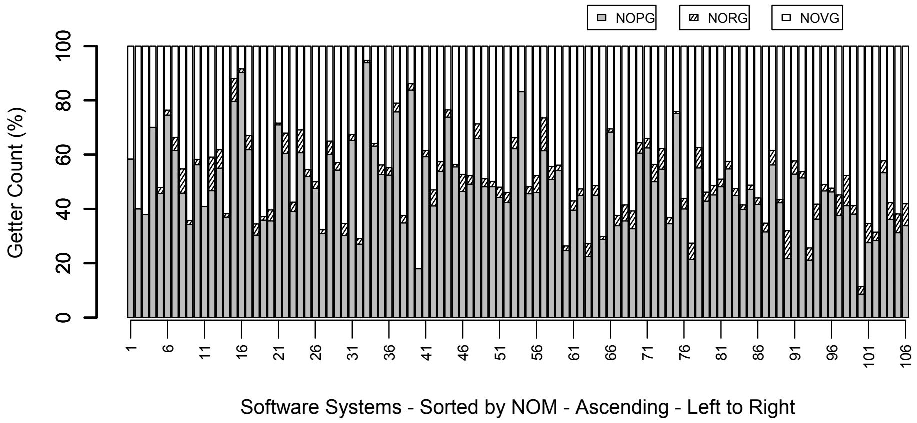


Figure 5.8: Proportion of Pure, Real, and Virtual Getters methods in software systems of Qualitas Corpus.

The proportions of their variants (i.e., real, virtual, pure) reveal additional insights into which specific type of getter and setter methods are being practiced more by developers. Figure 5.8 depicts that a substantial proportion of getter methods are pure and virtual in almost all the software systems in Qualitas Corpus. The proportion of pure getter methods (NOPG) ranges from 8.58% (compiere) to 93.85% (sablecc) with an inter-quartile range (IQR) of 36% to 56%. About 42% software systems employ more than 50% of their getter methods as pure getters. On the other hand, the proportion of virtual getter methods (NOVG) ranges from 5.2% (compiere) to 88.6%(sablecc) with IQR of 38 to 59%. About 49% software systems comprise more than 50% of their methods as virtual getters. Real getter methods (NORG) make up only small proportion (less than 13%) in all software systems in the Qualitas Corpus.

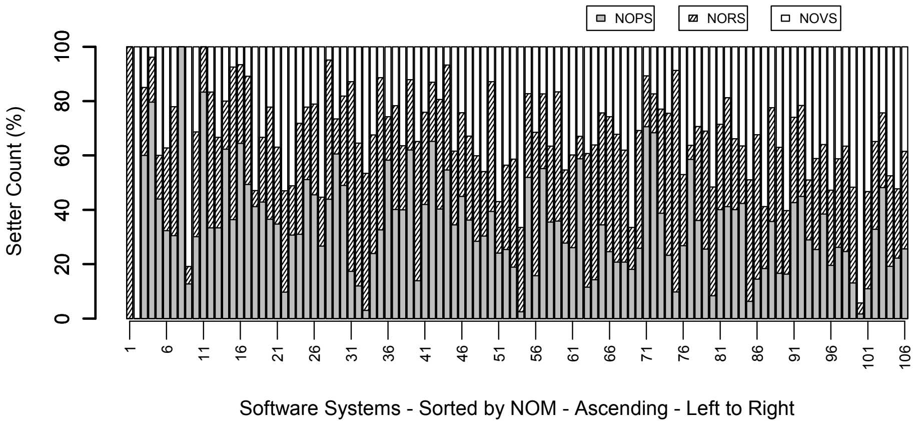
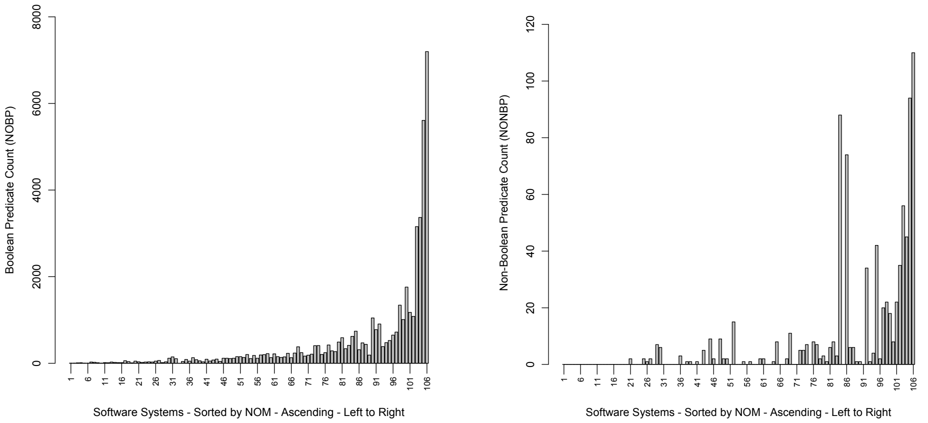


Figure 5.9: Proportion of Pure, Real, Virtual and Setters methods in software systems of Qualitas Corpus.

On the other hand, the real, virtual, and pure setter methods are employed at varying degree of proportions (cf. Figure 5.9). The proportion of pure setter ranges from 0% (jasml) to 100% (jparse) with an interquartile range (IQR) of 20% to 42%. Unlike real getter methods, the occurrence of real setter methods is substantially higher in number. Their proportions varies from 0% (fitjava) to 100% (jasml) with an IQR of 25% to 41%. Such high proportion of real setters suggests that developers usually implement some functionality before setting fields. The proportion of virtual setter methods ranges from 0% (jparse) to 100% (fitjava) with an IQR of 21.24% to 41.25%.

While developers employ getter and setter methods along with their variants to a varying degree, predicates (e.g., isTrue, isOpen) appear much more sparsely in the software systems of the Qualitas Corpus. Figure 5.10 depicts their actual occurrences in different software systems of Qualitas Corpus. Only two software systems of Qualitas Corpus comprise more than 5000 boolean predicates. These are netbeans and eclipse that comprise 5,609 and 7,195 boolean predicates, respectively. On the other hand, non-boolean predicates are very few in number

(<120). Netbeans and eclipse comprise the 94 and 110 non-boolean predicates, respectively. This finding suggests that developers, when defining predicates, return boolean values with some exceptions, and thus comply with conventions.



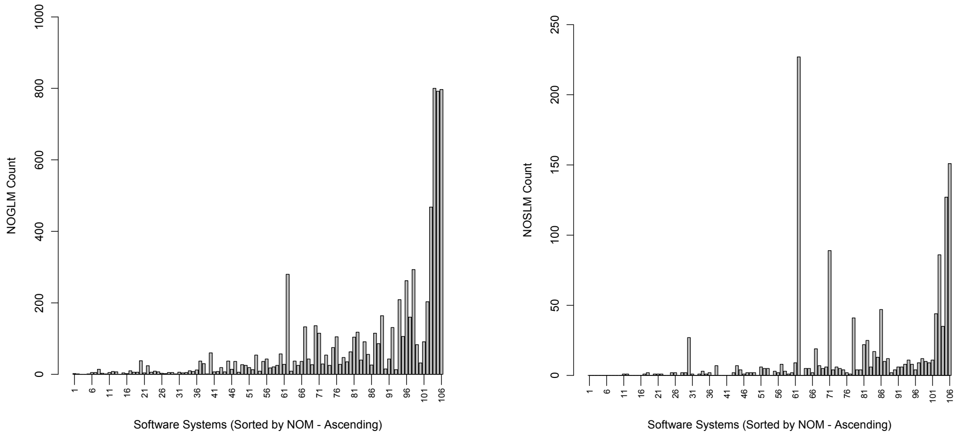
(a) Boolean Predicate

(b) Non-Boolean Predicate

Figure 5.10: Boolean and Non-Boolean predicates in the software systems of Qualitas Corpus.

Figure 5.11 depicts the actual occurrences of getter- and setter-like methods in the software systems of Qualitas Corpus. The proportion of getter-like methods in software systems of the Qualitas Corpus ranges from 0% (nekohtml) to 5.3% (roller). We observed that 76% software systems employ less than 1% getter-like methods. But in large software systems, such small proportion can still result in a high number of such methods (e.g., eclipse: NOGLM = 797, 0.5%, netbeans: NOGLM = 792, 0.58%, jre: NOGLM = 750, 0.78%, jboss: NOGLM = 468, 0.67%).

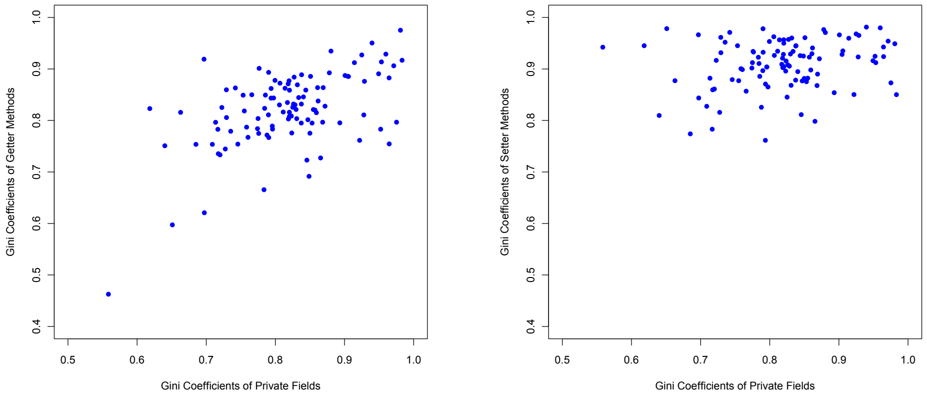
There are only a few setter-like methods (i.e., 0% to 4.3%) when compared to that of getter-like methods. We found the highest occurrence of setter-like methods in roller (NOGLM = 227, 4.3%). About 97% software systems including eclipse (NOSLM = 151, 0.09%) and netbeans (NOSLM = 127, 0.09%) comprise less than 1% setter-like methods.



(a) Getter-like Methods

(b) Setter-like Methods

Figure 5.11: Getter-like and Setter-like Methods in the software systems of Qualitas Corpus.



(a) Distribution of Private Fields and Getter Methods

(b) Distribution of Private Fields and Setter Methods

Figure 5.12: Distribution of Private Fields vs. Distribution of Getter and Setter Methods in Software Systems of Qualitas Corpus

5.3.3 Relation between Private Fields and Getter-Setter Methods

Do the developers use the property mechanism to circumvent the visibility modifiers, in particular, private? Do they define getter and setter methods when they define private fields by default? To answer these questions, we investigated the correlation between private fields and getter and setter methods.

We found that the property mechanism is not being used to circumvent the private visibility modifiers. The concentrations of getter and setter methods are different from that of private fields (cf. Figure 5.12). While the correlation (Spearman's ρ) between the distributions of getters methods and private fields is 0.54, the correlation (Spearman's ρ) between the distributions of setters methods and private fields is only 0.31. These substantially weak correlations do not support the hypothesis that developers always define getter and setter methods with private fields of a class (cf. RQ8).

The above results present the distribution of getter and setter methods, but does not inform us about any potential correlation between private fields and variants (i.e., pure, real, virtual) of getter setter methods. Among them, only the pure getter and setter methods are designated to retrieve and modify associated private fields. The other two are involved with some additional tasks. To know whether developers define getter and setter methods with private fields *just* to access or modify them, we further investigated the relation between private fields and pure getter and setter methods.

We found that there is a varying degree of correlation between them across different software systems of the Qualitas Corpus. The computed correlations are summarized as box plots (cf. Figure 5.13). The IQR of correlation between private fields and pure getters ranges from 0.36 to 0.62 - which suggests a weak relation between them. We observed a strong positive correlation (above 0.80) between private fields and pure getters in only 11.32% of software systems. On the other hand, the IQR of the correlation between private fields and pure setters ranges from 0.18 to 0.46 - which indicates an even weaker relation between them. Only 5.6% of software systems show a strong correlation (above 0.80) between private fields and pure setters.

Such low correlation coefficients between private fields and pure getter and setter methods suggest that developers do not always tend to accompany a private fields with pure getter and setter methods. There is no empirical evidence to support a claim that when developers define a private field they will also define a getter and setter method by default. Though getter and setter methods are used on a regular basis (all stud-

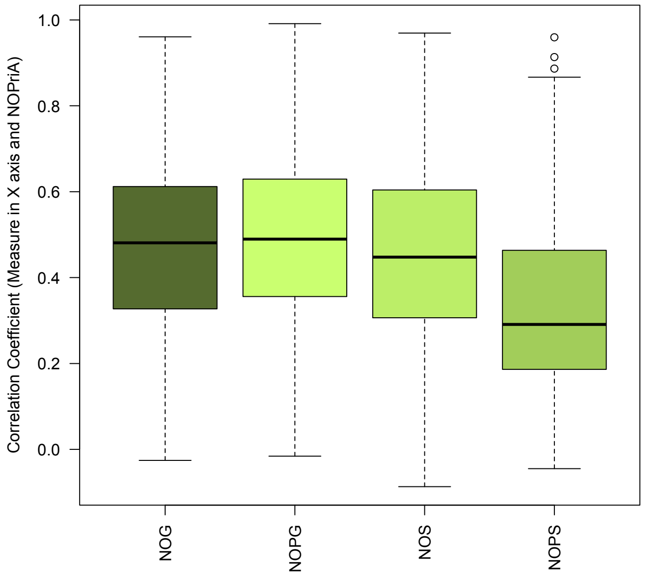


Figure 5.13: Boxplot of correlation coefficient between private fields and (pure) getter and setter methods in software systems of Qualitas Corpus.

ied systems make use of them), developers consciously employ them in a manner consistent with domain and system requirements.

5.3.4 Ratio of Getter and Setter Methods

What is the ratio between getter and setter methods? We found that if a class defines a getter method, then there is 36% chance that this class also defines a setter method, but the odds are not evenly distributed. Extending the ratio of getter and setter methods to 40% to 60% and 60% to 40%, respectively, yields a 55% probability that a getter method is accompanied by some setter method. Nevertheless, we found that the number of getter methods, in general, exceeds the number of setter methods by a factor of 2 to 1 in almost all systems of the Qualitas Corpus. Only ant-1.7.1 exhibits a dominance towards setter methods.

However, the dispersion of getter and setter methods is, in general, *inversely related* to their frequency (cf. Figure 5.14). Developers concentrate getter and setter methods in a few classes as much as possible. But as the proportion of these methods grows the need to disperse them also increases and becomes unavoidable eventually. Though this seems

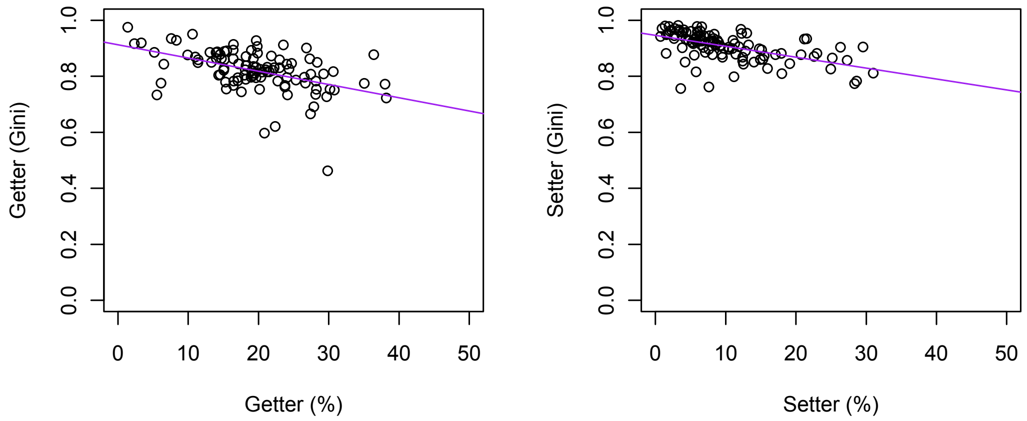


Figure 5.14: As the proportion of getter and setter methods increases, their concentration decreases though the correlation is not that strong (getter: Spearman ρ = -0.47, setter: Spearman ρ = -0.56)

natural, an important aspect of our finding is that even in cases where 30% of the methods are getters and setters, developers tend to centralize these methods in a few classes. The Gini coefficients for 30% of getter and setter methods assumes values typically greater than 0.7. This suggests that some level of bias towards model-separated design [76] is practiced.

5.4 Summary

In this chapter, we investigated how developers employ the property mechanism (i.e., getter and setter methods) in Java. Contrary to conventional belief, we found that these methods are neither commonplace nor “evil". Developers proactively select getter and setter methods in order to satisfy specific domain requirements.

Java lacks an appropriate built-in language support for property specification. As a consequence, even though Java’s “get" and “set" naming pattern can be used to develop software components (e.g., JavaBeans), it is merely a guideline that can be easily abandoned or misused.

We summarize the findings of this chapter below:

- There is no empirical evidence for intentional refactoring that would encapsulate fields by means of getter and setter methods.

- Given the considerable advice regarding whether to use or to avoid getter and setter methods (e.g., [69, 105, 106, 114, 183]), we found that developers use them extensively. Though getter and setter methods are used on a regular basis (most software systems in Qualitas Corpus make use of them), developers consciously employ them in a manner consistent with domain and system requirements.

- We found that the property mechanism is not being used to circumvent the visibility modifiers, in particular, private. There is no empirical evidence (in the Qualitas Corpus) to support a claim that when developers define a private field they will also define a getter and setter method by default.

- There is no empirical evidence (in the Qualitas Corpus) that the use of getter methods is a function of the solution design or domain. But, the same does not apply to setter methods. We observed that developers centralize data storage, but practice a much more decentralized and domain-independent approach to data retrieval.

- A getter method may not always be accompanied by a associated setter method. Our empirical evidence suggests that if a class defines a getter method, then there is an approx. 36% chance that this class also defines a setter method. The distribution of the ratio between getters and setter methods fits a normal distribution. In general, the number of getter methods exceeds the number of setter methods by a factor of 2 to 1 in any given system, though there are always a few classes in each system that contain more setter than getter methods. However, the proportion of getter and setter method is inversely related to their concentration.

Chapter 6

Inner Class Analysis

In this chapter, we focus on understanding developer behavior in using the notion of inner classes in Java. We set the stage for studying the use of inner classes at the beginning of this chapter. We then describe the analysis method in section 6.2. This comprises a definition of set the of software metrics and a description of the metrics analysis approach. We present the results of our investigation in section 6.3. We conclude this chapter by summarizing the key findings in section 6.4.

6.1 Introduction

The notion of inner class offers an abstraction mechanism that allows us to define a class in the context of another one, and thus provides us with a convenient means to structure intent (functionality) in program code. This concept, first introduced in Simula, is available in many object-oriented programming languages (e.g., C++, C#, Java, Ruby, and Python).

The Java programming language offers two types of inner classes[1]: static and non-static. The behavior of static inner classes is similar

[1] The term "inner classes" refers to nested types [84]. Though some literature (e.g., [157]) classify the nested types as static and non-static with the non-static nested classes called inner classes, we use the term *inner classes* to refer to all nested types in this chapter.

Table 6.1: Inner Classes in Java [84]

Type	Definition
Nested Top Level	Any type that is defined as a static member of another type. The behavior of this type is similar to that of top level types. This category includes the implicitly static types (i.e., nested interfaces, enumerated types, and annotation types).
Member	Any class that is defined as a non static member of an enclosing class. This category excludes the interfaces, enumerated types, and annotations as these are always implicitly static.
Local	A non static inner class that is defined within the body of a method, constructor or inside any static initializer block or instance initializer. The scope of this class is local to the hosting block. A local class has direct access to all the members in the hierarchy of enclosing classes (including private members), and also has access to final local variables and final method parameters of the enclosing method.
Anonymous	A local class that is nameless. An anonymous class cannot define any constructor as it has no name. Therefore, an *instance initializer* is used when a constructor is necessary.

to that of top level classes, except that it is nested inside another type. On the other hand, an instance of non-static inner class contains *implicit* and *hidden references* to instances of the associated outer classes. The non-static inner classes are divided into three types: Member, Local and Anonymous (cf. Table 6.1).

The implicit reference in Java's non-static inner classes offers a convenient means (i.e., callback) to implement specific types of functionality (e.g., GUI). For example, a software system comprising user interface components (e.g., button, menu) can employ non-static inner classes to accomplish an event handling mechanism [157, 195], and thus can utilize a callback facility to identify appropriate components that originate a particular event.

Besides, nesting a class inside another offers many other benefits [19]. These include logical grouping of classes (i.e., nesting *helper classes* assists to create more cohesive classes), better encapsulation (i.e., reduces the necessity to expose private members of host classes), and alternative to multiple inheritance (e.g., a super type of a nested class and its host class can be different).

Yet, inner classes (particularly anonymous ones) are often considered less powerful when compared to constructs like closures [126] - a feature available in other programming languages (e.g., Groovy). A closure is an anonymous chunk of code that can be passed around with the capability of accessing the original context.

Anonymous classes in Java, however, are considered somewhat similar to closures as both are attached to the enclosing environment. Though strictly controlled, an anonymous class can refer to selected variables (i.e., final variables in the enclosing method and all variables in the enclosing class) of the defining context. The underlying reason for such restricted access is associated with Java's approach to deal with local variables and parameters in methods. In Java, as usual, local variables are stored on the stack. The associated stack space is allocated when a method starts to execute and the stack is released when the method returns. But, the final variables are not stored on the stack, instead Java uses a method area to store them. Hence, final variables remain intact even after the method returns, whereas the non-final values are discarded (through the release of the associated stack space). Due to this restriction, Rose [184] identified anonymous inner classes as imperfect closures.

In addition to the closure aspect of inner classes, there is another dimension of debate that is concerned with their syntactic aspects. Anonymous classes in Java are considered more verbose and extremely clumsy as they have bulky syntax [184]. As a result, when used in program code, such classes can make resulting code somewhat difficult to read and thus can affect maintenance tasks. For this reason, many emphasize on making the syntax more concise. For example, Rose [184] suggests to replace such bulky syntax of inner classes with a more concise one.

```java
interface Runnable()
{
  public void abstract run();
}
```

Listing 6.1: A SAM Type

There exists a number of recommendations regarding inner classes and their alternatives. One of them is the Straw-Man proposal [182] that includes a guideline for replacement of anonymous classes with lambda expressions. Lambda expression targets SAM type (aka, *functional in-*

terface [18]) - an abstract class or interface that comprises only one abstract method. For example, Java's *Runnable* interface is a SAM type as it comprises only an abstract method *run()* (cf. Listing 6.1). An anonymous class that uses a SAM type can be be replaced by a lambda expression (cf. Listing 6.2).

```
//Anonymous class
Thread th = new Thread(new Runnable()
            {
              public void run()
              {
                CallMethodOne();
                CallMethodTwo();
              }
            })
//Corresponding lambda expression
Thread th = new Thread(#(){CallMethodOne(); CallMethodTwo();})
```

Listing 6.2: An anonymous class that uses a SAM type can be replaced by a lambda expression [182].

However, though there is much debate associated with the concept of inner classes, we do not have empirical evidence on how inner classes are actually being practiced by developers. We do not know yet to what extent developers employ inner classes while formulating a solution design and how those inner classes are being defined. For example, are most of the anonymous classes defined using SAM types? What is the typical type definition pattern of anonymous classes in particular and inner classes in general? Are they defined using inheritance?

The much blamed bulkiness of inner classes (that is considered to affect code readability) is not yet well-understood. Together with the syntax of inner classes, their size can also contribute to the bulkiness. One measure to gain an insight into their size is the number of methods hosted in inner classes. The more methods an inner class comprise, the more functionality it is concerned with, and consequently more bulky it can be. For example, an anonymous class (comprising a large number of methods) defined inside a method may contribute to bulkiness. Though number of methods in an anonymous class depends on the interface or the super class being implemented or extended, to what extent do developers employ such anonymous classes? What is the typical method distribution profile in anonymous classes in particular and inner classes in general?

Another aspect that one should also consider is the *nestedness* of inner classes. As nesting is considered to increase code complexity, highly nested inner classes may also affect code readability. Therefore, it is often advised to structure program code by avoiding deep nesting, and thus to yield better readability [95]. But how are inner classes nested in Java-based software systems? Do developers highly nest them, like Russian dolls, and thus create complex abstractions?

The answers to the above questions can provide us with better insights into the use-cases of the inner class mechanism in Java, and thus may assist the language change proposals (e.g., Straw-Man proposal [182]). We therefore investigated developer practices in using inner classes in Java-based software systems. It should be noted, however, that we did not attempt to justify the proposals, rather we demonstrated some of the important features of developer practices only. In particular, we addressed the following questions in this chapter.

- **RQ10** What is the typical distribution profile of inner classes in Java-based software systems? Does the use of inner classes vary across different domains (e.g., database, middleware)?

- **RQ11** Do developers confine the use of inner classes to few classes only?

- **RQ12** Do developers use inheritance substantially to define inner classes?

- **RQ13** Do developers create highly nested abstractions using inner classes?

- **RQ14** What is the typical distribution profile of methods in inner classes? Do SAM types occur with a high frequency?

6.2 Analysis Method

To answer our research questions, we defined a set of software metrics. We analyzed these metrics with different measures (e.g., Gini coefficient).

Table 6.2: Rationale for the Selected Key Measures

Name	**Rationale**
Inner Classes Distribution	Provides us with an insight into the actual extent of inner classes that developers employ while structuring solutions. Thus, it reveals the involvement of the inner class concepts in solution designs.
Inner Classes Hosting Classes	Reveals density of inner classes hosted in a class, and thus inform us whether developers employ inner classes in almost all the classes in a software systems or whether inner classes are defined in only a small proportion of classes).
Inheritance Structure of Inner Classes	Describes the extent to which the inner classes are influenced by the properties of their ancestors [56], and thus indicates to what extent developers rely on inheritance to define inner classes.
Degree of Nestedness of Inner Classes	Reveals level of nesting of inner classes, and thus indicates whether developers write complex abstractions (as represented by nesting level).
Breadth of Functional Decomposition in Inner Classes	Reveals structural semantics of inner classes, decomposed into number of methods. Thus it offers an indication of size of the associated inner class.

Definition of Metrics

To characterize the use of inner classes in Java-based software systems, we defined a set of software metrics that capture different aspects of developer practices in using inner classes.[2] In particular, we focused on 5 specific use-cases as presented in Table 6.2.

We defined corresponding metrics that capture different use-cases of inner classes. For example, we used the *depth in nesting level* (DNL) metric, similar to the *depth in inheritance tree* (DIT) metric [56], to quantify how deeply an inner class is nested. Like the DIT metric (i.e., the higher the value of DIT for a class is, the deeper its location in inheritance tree), the higher the value of DNL for an inner class is, the more nested it is. Consider, for example, a nesting structure of 3 classes: *A*,

[2] We considered how developers define inner classes, not how an instance of an inner class is used to access attributes in an instance of the outer (host) class.

B, and *C*. The class *A* contains class *B*, and the class *B* contains class *C*. In this case, the DNL values of these classes are 0, 1, and 2, respectively. Given a class, we counted its inner classes recursively (i.e., the class *A* has 2 inner classes and class *B* has 1 inner class).

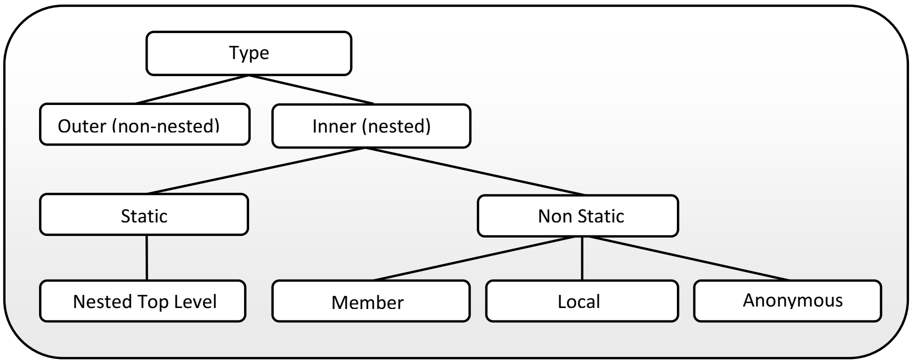


Figure 6.1: Classification of inner classes followed in our analysis.

Table 6.3: Measures Focused on Different Types of Inner Classes

Inner Class Measures		
Name	**Purpose**	**Relation**
Number of Types (NT)	Counts all defined classes and interfaces	-
Number of Outer Classes (NOC)	Counts all defined classes which are not enclosed in other classes	$NOC \subseteq NT$
Number of Declared Inner Classes (NDIC)	Counts all defined classes which are enclosed within other classes	$NDIC \subseteq NT$
Number of Declared Nested Classes (NDNC)	Counts all defined static nested classes	$NDNC \subseteq NDIC$
Number of Declared Member Classes (NDMC)	Counts all defined non static member classes	$NDMC \subseteq NDIC$
Number of Declared Local Classes (NDLC)	Counts all defined non static (and named) local classes	$NDLC \subseteq NDIC$
Number of Declared Anonymous Classes (NDAC)	Counts all defined non static anonymous classes	$NDAC \subseteq NDIC$

In Qualitas Corpus, the proportion of nested interfaces is very small (less than 3%). We decided not to study them in isolation. However, the definitions of all collected metrics are presented in Tables 6.3 to 6.7. We followed the classification depicted in Figure 6.1 while defining metrics.

Table 6.4: Measures Focused on Depth in Inheritance Tree of Inner Classes

Inner Class Measures		
Name	**Purpose**	**Relation**
Depth of Inheritance Tree	Depth in inheritance tree of a given inner class	-
Number of Inner Classes Defined Using Inheritance (ICDUI)	Counts all inner classes with DIT > 1	$ICDUI \subseteq NDIC$
Number of Nested Top Level Classes Defined Using Inheritance (NCDUI)	Counts all nested top level classes with DIT > 1	$NCDUI \subseteq NDNC$
Number of Member Classes Defined Using Inheritance (MCDUI)	Counts all member classes with DIT > 1	$MCDUI \subseteq NDMC$
Number of Local Classes Defined Using Inheritance (LCDUI)	Counts all local classes with DIT > 1	$LCDUI \subseteq NDLC$
Number of Anonymous Classes Defined Using Inheritance (ACDUI)	Counts all anonymous classes with DIT > 1	$ACDUI \subseteq NDAC$

Table 6.5: Measures Focused on Typical Parents (Interfaces / Super Classes) of Inner Classes

Parents (of Inner Classes) Measures		
Name	**Purpose**	**Relation**
Number of Parents of Inner Classes (NOPIC)	Counts all unique interfaces and super classes implemented and extended, respectively, by inner classes	-
Number of Library Interfaces and Super Classes (LISC)	Counts total number of unique interfaces/super classes that belong to the Java standard library [17]	$LISC \subseteq NOPIC$
Number of Other Interfaces and Super Classes (OISC)	Counts total number of unique interfaces/super classes that do not belong to the Java standard library	$OISC \subseteq NOPIC$
Number of Implementations/Extensions of Library Interfaces and Super Classes (IELISC)	Counts total number of times the LISCs are implemented/extended by inner classes	-
Number of Implementations/Extensions of Other Interfaces and Super Classes (IEOISC)	Counts total number of times the OISCs are implemented/extended by inner classes	-

Table 6.7: Measures Focused on Methods in Inner Classes

Method (in Inner Class) Measures		
Name	**Purpose**	**Relation**
Number of Methods in All Classes (NOMALLC)	Counts total number of non-abstract methods defined in all classes (i.e., outer, inner)	-
Number of Methods in Outer Classes (NOMOC)	Counts total number of methods defined in outer classes	$NOMOC \subseteq NOMALLC$
Number of Methods in Nested Top Level Classes (NOMNC)	Counts total number of methods defined in nested top level classes	$NOMNC \subseteq NOMALLC$
Number of Methods in Member Classes (NOMMC)	Counts total number of methods defined in member classes	$NOMMC \subseteq NOMALLC$
Number of Methods in Local Classes (NOMLC)	Counts total number of methods defined in local classes	$NOMLC \subseteq NOMALLC$
Number of Methods in Anonymous Classes (NOMAC)	Counts total number of methods defined in anonymous classes	$NOMAC \subseteq NOMALLC$

Table 6.6: Measures Focused on Depth in Nesting Level of Inner Classes

Inner Class Measures		
Name	**Purpose**	**Relation**
Depth in Nesting Level	Depth in nesting level of a given inner class	-
Number of Inner Classes Nested at Level Two or More (ICNLTM)	Counts all inner classes with DNL > 1	$ICNLTM \subseteq NDIC$
Number of Nested Top Level Classes Nested at Level Two or More (NCNLTM)	Counts all nested top level classes (and interfaces) with DNL > 1	$NCNLTM \subseteq NDNC$
Number of Member Classes Nested at Level Two or More (MCNLTM)	Counts all member classes with with DNL > 1	$MCNLTM \subseteq NDMC$
Number of Local Classes Nested at Level Two or More (LCNLTM)	Counts all local classes with with DNL > 1	$LCNLTM \subseteq NDLC$
Number of Anonymous Classes Nested at Level Two or More (ACNLTM)	Counts all anonymous classes with DNL > 1	$ACNLTM \subseteq NDAC$

Data Analysis Approach

We used the same measures (e.g., the Gini coefficient) that we employed in previous chapters. We followed frequency distribution analysis technique in this chapter too. For example, we used box plots to summarize proportion of different inner classes (e.g., member, local), and bar plots to depict their system specific proportions.

6.3 Observations

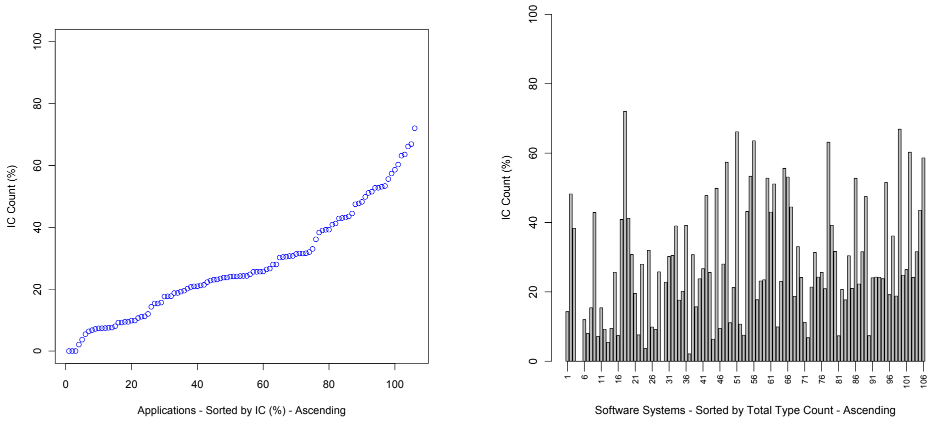
In this section, we present the result of our empirical investigation of different aspects of usage patterns of inner classes in Java-based software systems. This section is divided into: (i) inner class distribution, (ii) inner class distribution across different domains, (iii) density of inner classes in enclosing classes, (iv) inheritance structure of inner classes, (v) degree of nestedness of inner classes, and (vi) breadth of functional decomposition in inner classes.

6.3.1 Inner Class Distribution

To what extent do developers employ inner classes while organizing solution design? To answer this question, we investigated the proportion of different inner classes in the software systems of Qualitas Corpus. In this section, we present the resulting outcomes. In particular, we focus on (i) degree of share of inner classes in the total class space, and (ii) degree of share of different types of inner classes (e.g., nested top level, member, local and anonymous) in total class space.

Degree of Share - Overview

The amount of inner classes in the software systems of the Qualitas Corpus ranges from 0% to almost 75% of the total system size (measured in terms of number of types). While Figure 6.2(a) depicts the distribution of share of inner classes in different software systems, Figure 6.2(b) indicates that the use of inner classes in a solution design does not depend on the size of the resulting software system. Developers employ inner classes as they see fit.



(a) (b)

Figure 6.2: Proportion of Inner Classes in the Investigated Software Systems (a) Sorted (ascending) based on proportion of Inner Classes (b) Sorted (ascending) based on total (outer and inner) *Type Count*

Yet, the use of inner classes in a software system can depend on variety of factors. These include system specific requirement and developer design choices. In addition, the underlying problem domain of a software system can also be a contributing factor to motivate developers to formulate a solution design using inner classes. For example, developers of a software system that requires GUI functionality are likely to prefer inner classes to utilize the benefits of the callback mechanism offered by non-static inner classes.

Table 6.8: Least inner classes employing software systems (7% of Qualitas Corpus).

System	Size	Inner Classes (%)	Description
jFin_DateMath	60	0	Trade processing utility.
jparse	69	0	XML parser.
mvnforum	272	0	J2EE technology(Jsp/Servlet) based bulletin board.
ivatagroupware	378	2	J2EE technology based groupware solution.
jrat	246	4	Runtime performance profiler for the Java platform.
freecs	147	5	Java-based chat server.
c_jdbc	581	6	Database middleware.

But to what extent do software systems employ inner classes? What are the reasons? We observed that the software systems utilizing either no or only a few inner classes (cf. Table 6.8) are rather small in size. Their system size (in terms of type count) ranges from 60 (jFin_DateMath) to 581 (c_jdbc).

Table 6.9: Most inner classes employing software eystems (7% of Qualitas Corpus).

System	Size	Inner Classes (%)	Description
jomeny	193	72	Plugin-based Accounting Framework
drjava	3866	67	Lightweight Development Environment for Java
gallon	803	66	Media Server
pooka	869	64	Email Client
rssowl	1682	63	News Feed Reader
azureus	7231	60	Bittorrent Client
netbeans	32475	59	IDE

On the other hand, software systems that make extensive use of inner classes (cf. Table 6.9) have varying degrees of system size, ranging from 193 (jomeny) to 32,475 (netbeans). Most of the inner class-rich software systems incorporate an extensive amount of GUI functionality. For example, galleon - a media server, employs inner classes for implementing GUI components. But an extensive use of inner classes is not just restricted to GUI functionality. For example, jmoney - an accounting framework, is also an inner class-rich software system that uses a plug-in architecture (e.g., user interface components, new properties to accounts, transactions). Another example is drjava - a lightweight development environment for Java. This software system employs inner classes for implementing interaction interpreters (particularly for the beginners) that allows quick evaluation of Java expressions.

Table 6.10: Proportion of different *inner class intensive* software systems

Proportion of Different Inner Class Intensive Software Systems		
Name	**Definition**	**Count (%)**
Inner Class Intensive	Counts software systems comprising more than 50% inner classes of total classes	15
Nested Top Level Class Intensive	Counts software systems comprising more than 50% nested top level classes of total inner classes	27
Member Class Intensive	Counts software systems comprising more than 50% member classes of total inner classes	7
Local Class Intensive	Counts software systems comprising more than 50% local classes of total inner classes	0
Anonymous Class Intensive	Counts software systems comprising more than 50% anonymous classes of total inner classes	36

Degree of Share - Different Inner Classes

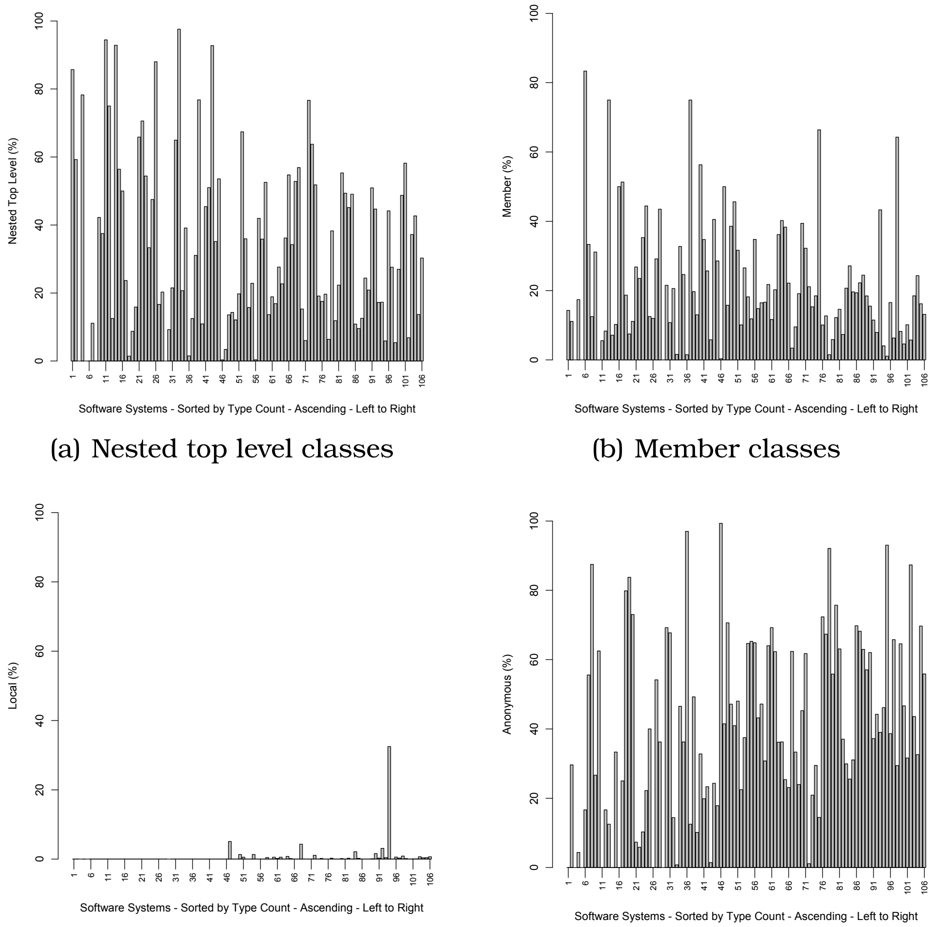
What are the contributions of different types of inner classes (e.g., local, anonymous, member, nested top level) in total class space? While Figure 6.3 depicts the proportion of share of various inner classes, Table 6.10 reveals the proportion of software systems comprising more than 50% inner classes. We describe the usage profiles of different inner classes in various software systems below.

Nested Top Level Classes

The software systems in the Qualitas Corpus comprise a varying proportion of nested top level classes (cf. Figure 6.3(a)), ranging from 0% to 98%. The highest proportion (98%) is observed in emma, a code coverage tool. The purpose of using extensive nested top level classes in this software system includes the implementation of visitor patterns [85], html report generation, and parsing.

Besides emma, there are 5 more software systems in the Qualitas Corpus that employ an extensive amount of nested top level classes. These are cobertura, jasml, itext, javacc, and velocity. They comprise more than 80% nested top level classes. It is important to note that these software systems are small in size (median system size is 201 types).

On the other hand, we observed that 38% of the software systems define less than 20% of their total inner classes as nested top level classes. This includes 2 software systems (i.e., quilt and webmail) that do not employ any nested top level classes at all.



(a) Nested top level classes

(b) Member classes

(c) Local classes

(d) Anonymous classes

Figure 6.3: Proportion of various inner classes. (Percentage is calculated with respect to all inner classes of given software system. For example, percentage of anonymous classes in a software system is computed based on total inner classes in that software system.)

Member Classes

Unlike instances of nested top level classes, an instance of a member class has an implicit reference to an instance of its enclosing class. Developers may utilize that implicit reference through the use of member classes. But empirical evidence suggests that such classes are not being used extensively in most of the software systems. More than half (56%) of the software systems in the Qualitas Corpus contain less than 20% member classes only (cf. Figure 6.3(b)). The proportion of *member class intensive* applications is relatively small (7%) compared to other types of inner classes (cf. Table 6.10).

Local Classes

Figure 6.3(c) shows that the use of local classes[3] is very rare in the studied software systems. In Qualitas Corpus, 67% of the software systems do not use local class at all. On the other hand, only 1 local class is found in 11% of the software systems, and 2 to 20 local classes are observed in 17% of the software systems. Only 5 software systems (i.e., netbeans, nakedobjects, jre, jboss, eclipse) employ more than 20 local classes, yet the proportion is less than 1% for all of these systems.

The underlying reasons of such limited use of local classes can be attributed to a developer design choices. Either they do not find the concept of local inner classes useful in formulating a solution design or they use some other language construct that they are more comfortable with. Whatever the reasons are, the rare use of the concept of local class indicates a possible limited utility in Java-based software development, which in turn merits its removal (or deprecation) from the Java language.

Anonymous Classes

Unlike local classes, developers make extensive use of anonymous classes (cf. Figure 6.3(d)). About 36% of the software systems in the Qualitas Corpus comprise more than 50% anonymous classes. These include 7 software systems (i.e., azureus, colt, ireport, jag, joggplayer, rssowl, webmail) where more than 80% of their total inner classes are anonymous.

The reason for such an extensive use of anonymous inner classes include a wide variety of tasks implemented by them. For example, azureus, a bittorrent client, uses anonymous classes for event management activities (e.g, connection management, disk management, packet handling, URI handling). Colt - a set of open source libraries that facilitate high performance scientific and technical computing in Java, employs anonymous classes for mapping of different types (e.g., int-

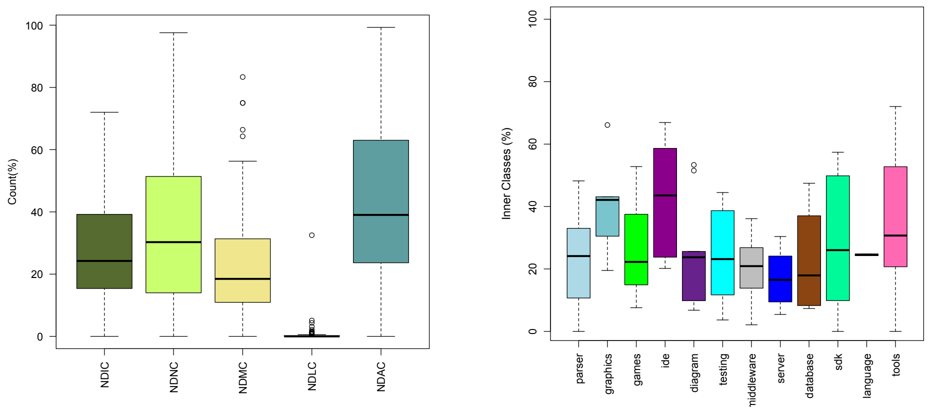
[3] Though anonymous classes are also local classes, with local classes we refer to the non-anonymous local classes only (i.e., local classes that do have an explicit name).

double, long-object). Another software system jag (Java application generator), employs anonymous classes for GUI functionality.

On the other hand, there are software systems that make limited use of anonymous classes. About 17% of the software systems use less than 10 anonymous classes. These include 5 software systems (i.e., cobertura, informa, jasml, javacc, velocity) that do not employ any anonymous inner classes at all. These systems are fairly small in size (i.e., median system size is less than 150). In addition, more than 80% of the inner classes are nested top level classes in all these systems (except the software system informa that comprises only 50% nested top level classes).

6.3.2 Inner Class Distribution - Domain Perspective

Given the varying degree of proportion of different inner classes in Java-based software systems, what are their proportions in different domains? Do the use of inner classes vary across domains (e.g., parser, middleware, tools)? To answer these questions, we studied the inner class usage profiles from a domain perspective. Our investigation reveals that the proportion of inner classes varies from domain to domain (cf. Figure 6.4).

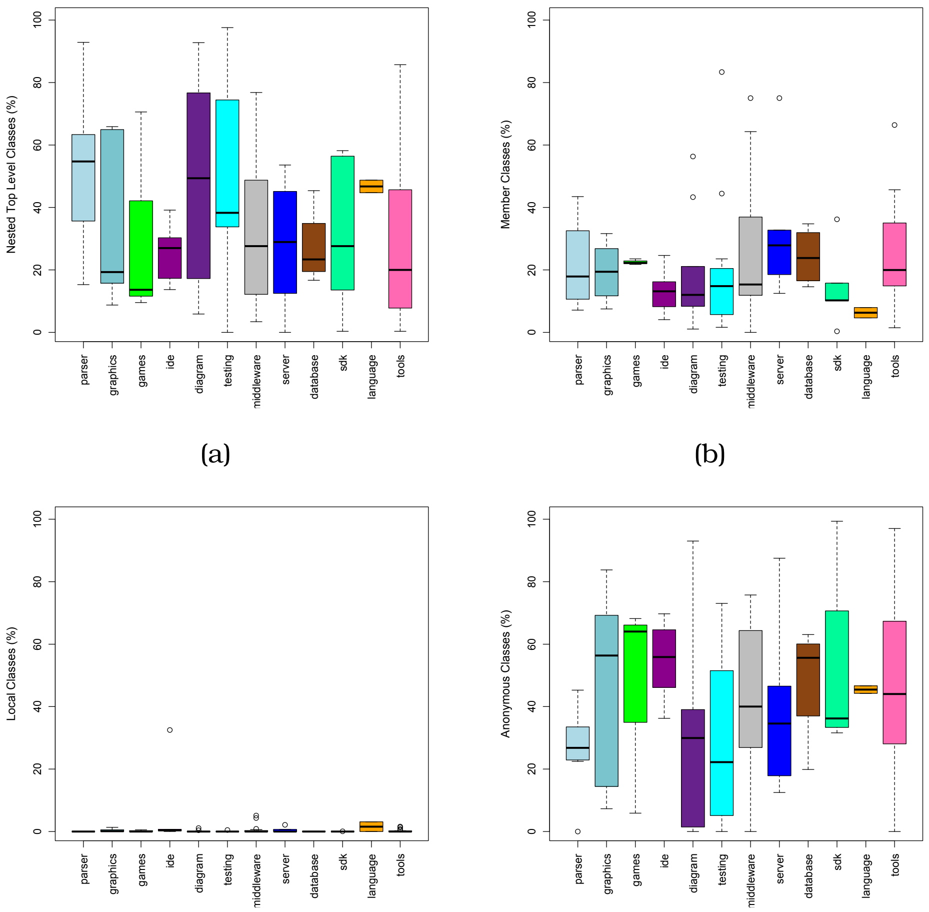


(a)

(b)

Figure 6.4: (a) Proportion of various inner classes (b) Proportion of inner classes in different domains

But what are the contributions of different inner classes (e.g., nested top level, member, anonymous, local) in various domains? To facilitate our understanding on which specific type of inner classes constitute more in a particular domain, we studied the domain-specific usage profile for each category of inner classes (cf. Figure 6.5).



(a)

(b)

(c)

(d)

Figure 6.5: Proportion of different inner classes in various domains (a) Nested Top Level Classes, (b) Member Classes, (c) Local Classes, (d) Anonymous Classes

We observed that anonymous classes are used substantially in most domains (7 domains comprise more than 40% anonymous classes - cf. Table 6.11). Only 3 domains (i.e., parser, diagram generator and language) use nested top level classes over 40%. It is evident that the use of local classes is very limited in almost all domains (cf. Figure 6.5(c)).

Table 6.11: Median of proportion of different inner classes in various domains

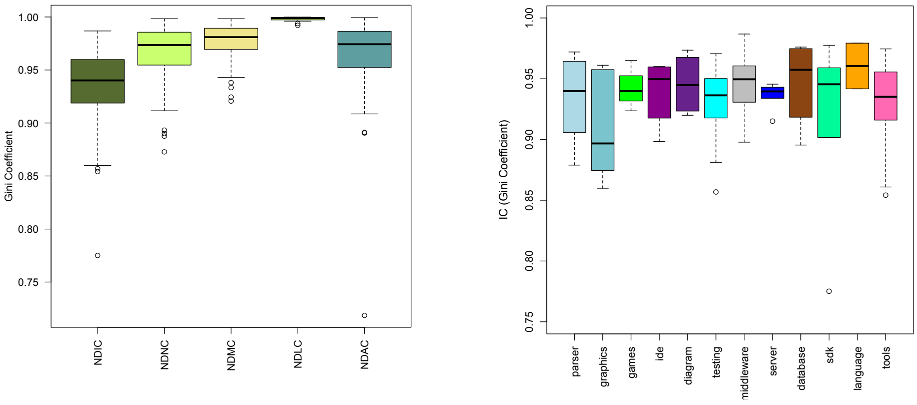
Median (%)					
Domain	**Inner Classes**	**Nested Top Level Classes**	**Member Classes**	**Local Classes**	**Anonymous Classes**
Parser	24.14	**52.56**	16.66	0.00	23.97
Graphics	**42.12**	19.32	19.41	0.11	**56.34**
Games	22.27	13.65	22.25	0.00	**64.02**
IDE	**43.55**	26.99	13.15	0.42	**55.86**
Diagram	23.75	**49.36**	12.00	0.00	29.94
Database	17.92	23.35	23.81	0.00	**55.61**
SDK	26.03	20.59	10.20	0.00	34.76
Middleware	20.91	27.61	15.33	0.00	**40.00**
Server	16.51	28.94	27.86	0.00	34.56
Language	24.54	**46.71**	6.28	1.56	**45.44**
Testing	23.14	**38.29**	14.81	0.00	22.22
Tools	30.70	19.12	19.69	0.00	**40.94**

While the IQRs of nested top level and anonymous classes covers a broad range in most of the domains, the IQRs of member classes are comparatively confined within a small range (10% to 30%).

But what are the reasons for such varying proportions of different inner classes in various domains? The underlying reasons include the nature of functionality implemented in the constituent software systems. For example, jhotdraw, a software system in the graphics domain, uses anonymous classes for implementation of GUI and drawing functionality. Antlr, a software systems in the parser domain, uses nested top level classes for grammar specification in a parser class.

6.3.3 Density of Inner Classes in Enclosing Classes

While the proportion of inner classes discussed above illustrates the actual scenario that captures to what extent developers employ different types of inner classes, it does not inform us how many classes host inner classes. Do developers formulate solution designs by defining inner classes in a few number of enclosing classes or do they disperse them across the classes of a software system?



(a)

(b)

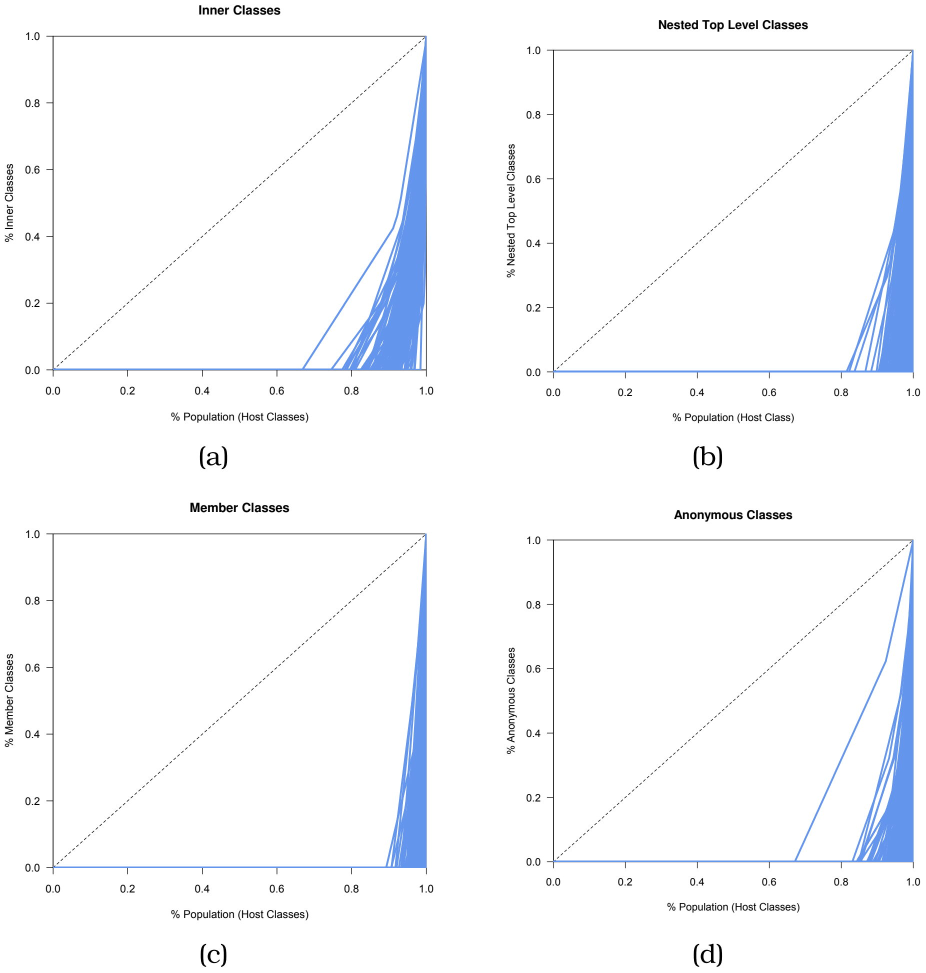
Figure 6.6: (a) Concentration profile of various inner classes (b) Concentration profile of inner classes in various domains

The distribution of inner classes in the investigated software systems is highly concentrated in only a few *top-level* classes. The Gini coefficient of inner classes in 93% of the software systems of the Qualitas Corpus is above 0.88, suggesting a highly uneven distribution of inner classes. This observation holds even in cases when a software system make extensive use of inner classes. For example, though the software system eclipse has 41% inner classes, the Gini coefficient of inner classes is very high (0.92), suggesting that all the inner classes are defined in only a few top-level classes.

We observed that the width of the interquartile range (IQR) (cf. Table 6.12) is very narrow, suggesting a small band of inner classes' typical bounded regions. All the values are above 0.90, which indicates a highly concentrated distribution profile of inner classes. The interquartile range (IQR) of the *Number of Declared Inner Classes* distribution is the widest one. On the other hand, the interquartile range (IQR) of the *Number of Declared Local Classes* distribution is almost close to zero. This is expected as most of the software systems do not comprise any local class at all, limited use of local classes in several systems is restricted to few enclosing classes (discussed in previous sections). As a result, we observed a highly concentrated distribution (i.e., Gini coefficient above 0.99) and a very narrow interquartile range (i.e., width of IQR is 0.02) of local classes.

Measure	**Interval**	**Width**
Number of Declared Inner Classes	[0.918, 0.959]	0.041
Number of Declared Nested Classes	[0.955, 0.985]	0.030
Number of Declared Member Classes	[0.969, 0.989]	0.019
Number of Declared Local Classes	[0.997, 0.999]	0.002
Number of Declared Anonymous Classes	[0.952, 0.986]	0.034

Table 6.12: Narrow Bounded regions of inner classes (cf. Figure 6.7).



(a)

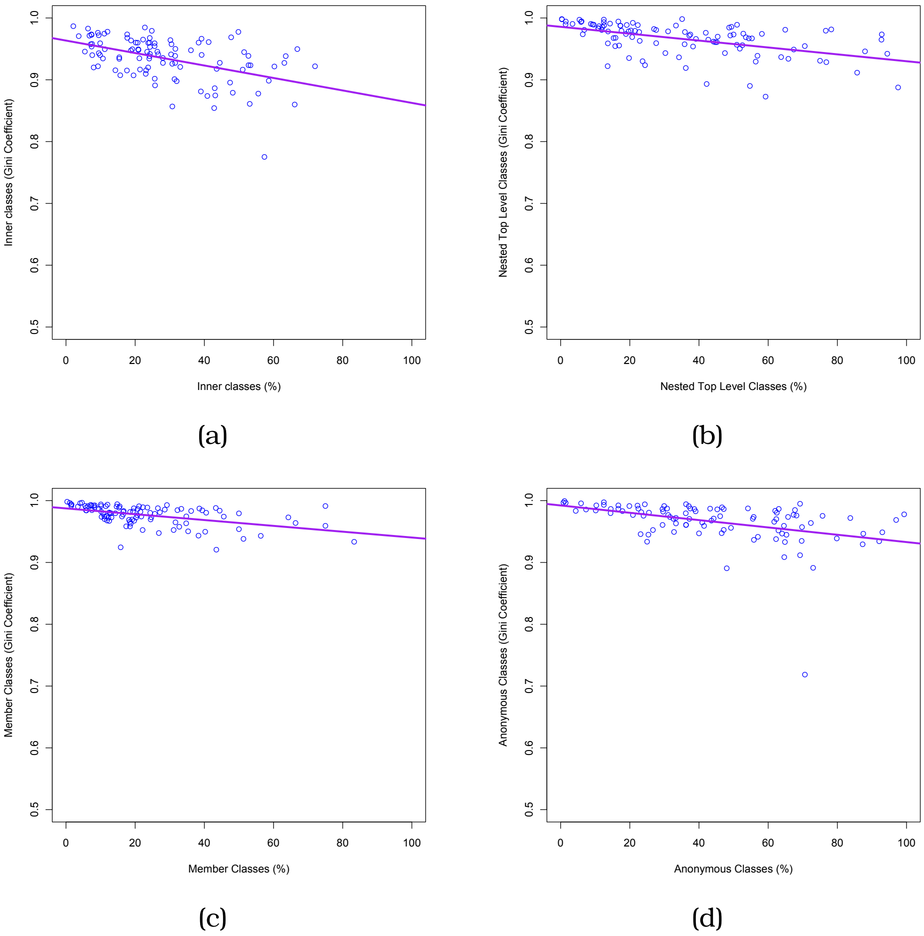
(b)

(c)

(d)

Figure 6.7: Narrow bounded regions of various inner classes (a) Inner classes (b) Nested top level classes (c) Member classes (d) Anonymous classes (Local classes are excluded due to very few data points)

Though developers define inner classes in a few number of top level classes, their concentration is *inversely* related to their occurrences. That is, if the total number of inner classes increases, the concentration of the inner classes decreases. Figure 6.8 shows that there is a *negative* relation between the proportion of inner classes present in a software systems and their corresponding concentration. The more inner classes are in Java-based software systems, the more they are dispersed (i.e., thus reducing the chance of growing inner class based *God-like* classes).



(a)

(b)

(c)

(d)

Figure 6.8: Relation between the proportion and distribution of different classes in the software systems of the Qualitas Corpus. (a) Inner classes, (b) Nested top level classes, (c) Member classes, and (d) Anonymous classes. (Local classes are excluded due to very few data points.)

6.3.4 Inheritance Structure of Inner Classes - Nature of Type Definition

When classes are defined using inheritance, many super classes can potentially influence them [55]. The deeper the class in the hierarchy, the more methods and classes are involved, resulting in a comparatively more complex solution design [55, 192]. Moreover, a higher depth in inheritance is often blamed for an increasing difficulty to maintain a software system [131]. For such reasons, substantial use of inheritance in object-oriented software systems is discouraged by many (e.g., [85]). Yet, developers may use inheritance for different reasons (e.g., code reuse).

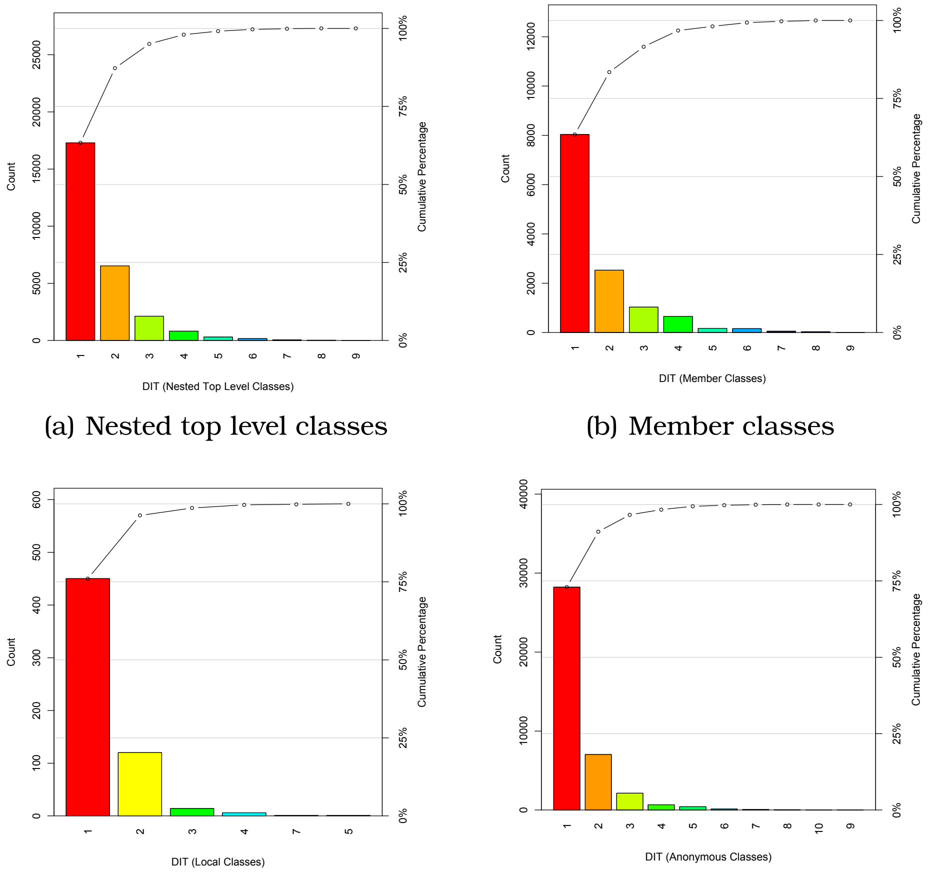
But how are inner classes being defined? Do developers use inheritance substantially to define inner classes? Do developers follow similar inheritance structure in case of both inner and outer classes? To what extent the developers use the Java standard library types [17] as parents of inner classes?

In this section, we present (i) Extent of inner classes defined using inheritance, (ii) Comparison of inheritance structure of inner and outer classes, and (iii) Use of system specific and standard library types as parents of inner classes.

Extent of Inner Classes Defined Using Inheritance

As depicted in Figure 6.9, a substantial proportion of inner classes in all the software systems of the Qualitas Corpus do not use inheritance. The value of DIT of most of the classes is 1.[4] This suggests that developers tend to somewhat limit the use of inheritance in inner classes (cf. Figure 6.9). When inner classes are defined using inheritance, most of them appear at level 2 in the inheritance tree. Yet, the highest values of DIT of different types of inner classes are quite high. We found that the highest value of DIT of both nested top level classes and member classes is 9, local classes is 5, and anonymous classes is 10.

[4] In our analysis, DIT = 1 means inherited from default parent `java.lang.Object`.



(a) Nested top level classes

(b) Member classes

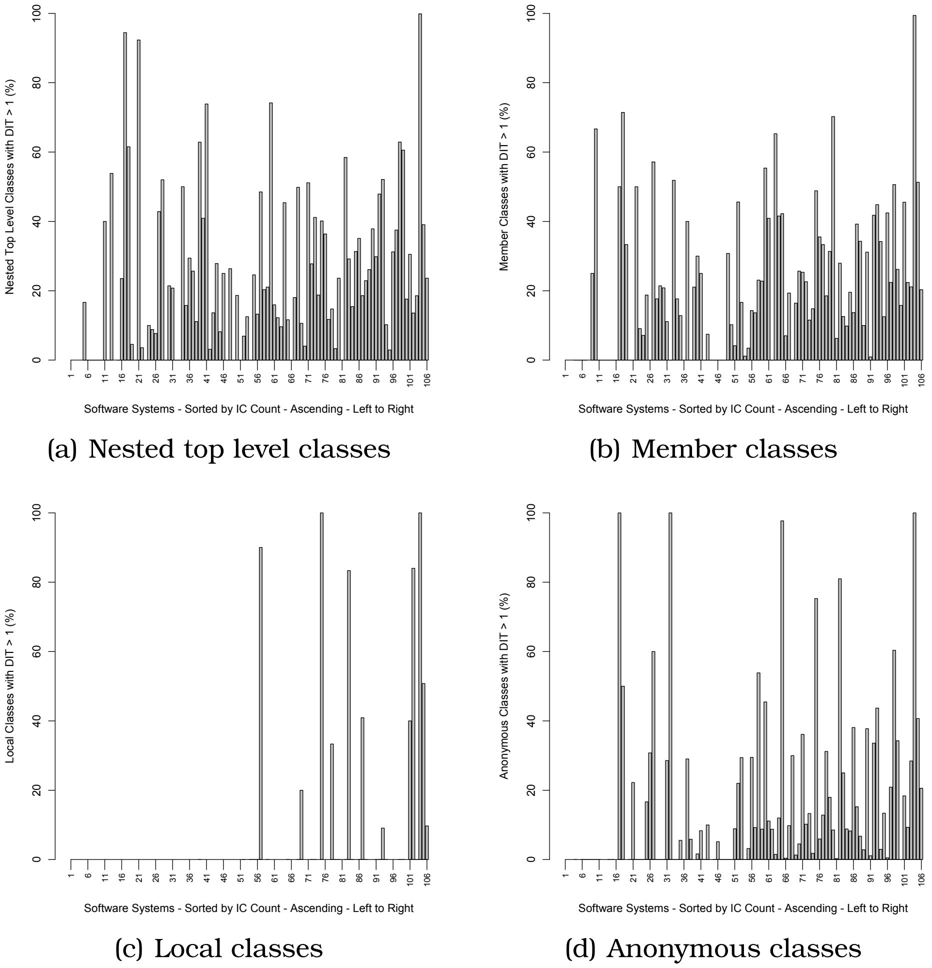
(c) Local classes

(d) Anonymous classes

Figure 6.9: DIT of various inner classes (Sorted from higher to lower count)

Figure 6.9 provides us with an overview of use of inheritance in inner classes in the entire corpus. In order to gain an insights into inheritance at individual software system level, we investigated the proportion of inner classes being defined using inheritance for each software system in Qualitas Corpus.

As shown in Figure 6.10(a), most of the software systems in the Qualitas Corpus have a limited proportion of nested top level classes that are defined using inheritance (NCDUI). We found 49% of the software systems that comprise less than 20% NCDUI. On the other hand, few software systems (i.e., c_jdbc, fitjava, jre) exhibit more than 90% NCDUI. But such high proportion is observed in some systems due to the use of only a few nested top level classes, For example, c_jdbc and fit-



(a) Nested top level classes

(b) Member classes

(c) Local classes

(d) Anonymous classes

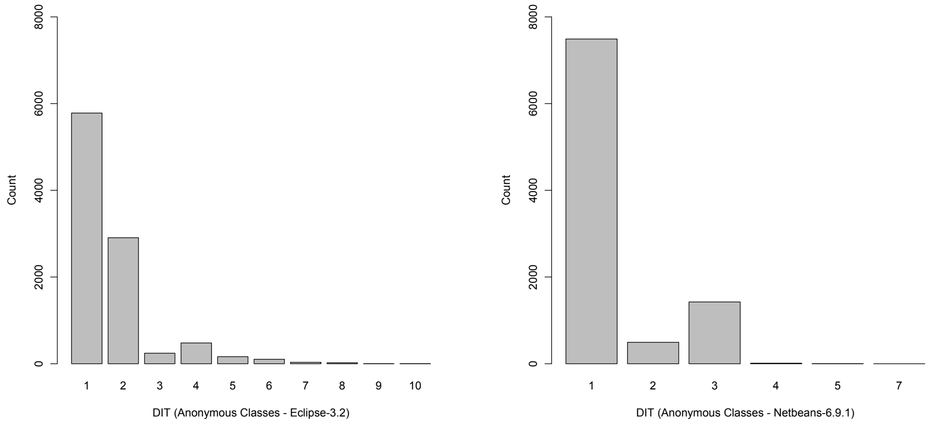
Figure 6.10: Inner classes that are defined using Inheritance (Percentage is calculated with respect to corresponding classes. For example, percentage of DIT>1 in anonymous classes of a given software system is computed based on total anonymous classes in that software system.)

java have less than 20 nested top level classes. The other system (jre) employs 2902 nested top level classes, and almost all of them (99%) are are NCDUI.

About 52% of the software systems comprise less then 20% member classes that are defined using inheritance (MCDUI) (cf. Figure 6.10(b)). The highest proportion (99%) of such classes is observed in jre (that has 5,397 member classes). Eclipse (with 2,264 member classes) has 51% MCDUI and Netbeans (with 2,504 member classes) has only 20% MCDUI.

Only 12% of the software systems define local classes using inheritance (LCDUI) (cf. Figure 6.10(c)). Though some software systems show a high proportion of LCDUI (e.g., 51% in eclipse, 100% in jre), they comprise only less than 35 LCDUI.

About 63% of the software systems define less than 20% anonymous classes using inheritance (ACDUI) (cf. Figure 6.10(d)). Yet, we observed 5 software systems that define more than 80% ACDUI. Some of these software systems employ only a few anonymous classes. These are fitjava (AC=1, ACDUI=100%) and sablecc (AC=25, ACDUI=100%). The rests employ a substantial number of anonymous classes. These are openjms (AC=321, ACDUI=81%), castor (AC=215, ACDUI=98%), and jre (AC=1,161, ACDUI=100%).



(a) Eclipse

(b) Netbeans

Figure 6.11: DIT of anonymous classes in two large software systems

Though limited use of inheritance is observed in defining inner classes, few software systems (e.g., jruby, eclipse) comprise inner classes that are located deeply (up to 10) in inheritance tree (cf. Figure 6.11). In eclipse, the purpose of using inner classes with higher DIT values include UI (user interface) functionality implementation. In jruby, we observed that inner classes with higher DIT values are the results of both machine generated code and written program code (cf. Figure 6.12).

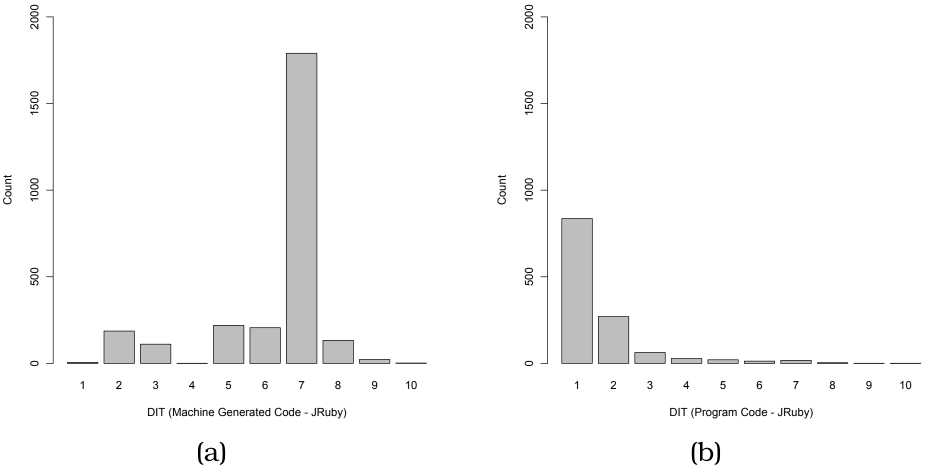


Figure 6.12: DIT of Inner classes in JRuby (a) Machine generated code, (b) Written program code

Comparison of inheritance structure of inner and outer classes

To gain an overview of the use of inheritance in defining outer and inner classes, we computed the proportion of both type of classes at each level of inheritance (e.g., DIT = 1, 2) for all the software systems of the Qualitas Corpus. The results are depicted in Figure 6.13.

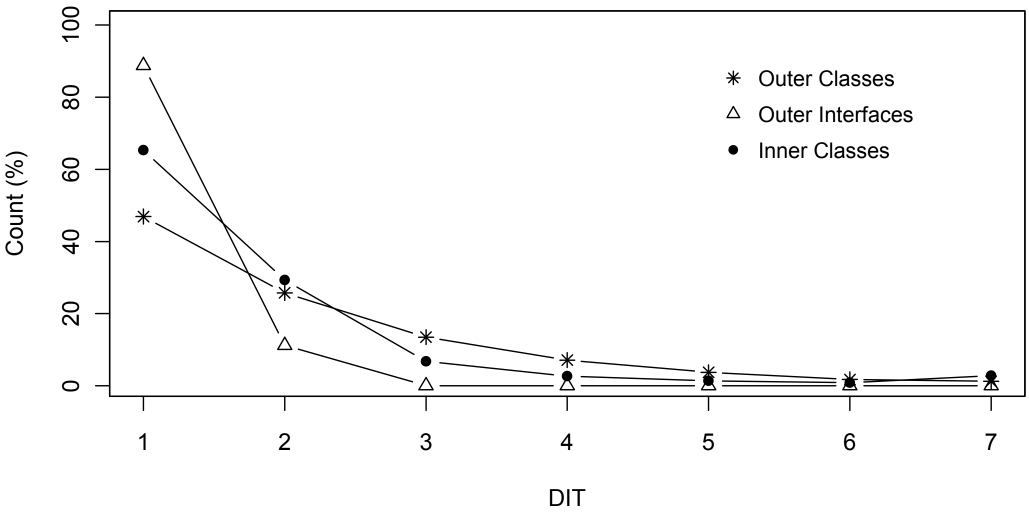


Figure 6.13: Proportion of different types (i.e., Outer classes, Outer Interfaces and Inner classes) with varying DIT values (DIT = 7 means DIT 7 and above) of the software systems in the Qualitas Corpus.

We observed that the proportion of inner classes (65% at DIT = 1, 29% at DIT = 2) are comparatively higher than that of outer classes (47% at DIT = 1, 26% at DIT = 2) at inheritance level one and two. At deeper levels (i.e., DIT = 3 and more), the proportion of outer classes dominate the proportion of inner classes. This suggests that more outer classes are involved in inheritance at higher depth.

Our observation, with regards to the use of inheritance in outer classes, supports the finding of a recent study, conducted by Tempero et al. [209], *"an apparently high use of inheritance is a characteristic of accepted Java programming practice"*. This study, however, did not consider inner classes, and therefore, the finding does not cover inner classes. We observed a limited use of inheritance in inner classes. As the purpose of inner classes include assisting functionality implementation in outer classes (i.e., acting as helper classes), the responsibility of both type of classes may not be exactly the same. Therefore, though high use of inheritance is observed in case of outer classes, the inner classes in the software systems of the Qualitas Corpus exhibit somewhat less use of inheritance.

Use of system specific and standard library types as parents of inner classes

What are the super classes and interfaces that the inner classes extend and implement? Are all of them belong to the Java standard library? The answer will assist us to gain an insight into typical parents (root) of inner classes, and thus can provide an indication of functionality being implemented by inner classes. Moreover, we can identify the extent of use of standard library in defining inner classes.

Though developers may define custom super classes and interfaces (and also use third party utility) based on system specific requirements, they may also make use of Java standard library [17] for a particular purpose. While the custom super classes and interfaces are likely to be system specific, the Java standard library appears to be a more common utility that can be used, in general, in different software systems.

Table 6.13: 20 Most Frequent Interfaces (Library) implemented by the Inner Classes, (These are 4% of the total interfaces but contribute 84% of the total implementations)

Interface	Count (%)
java.lang.Runnable	21.18
java.awt.event.ActionListener	16.45
java.security.PrivilegedAction	11.16
java.io.Serializable	6.10
java.util.Comparator	4.56
java.util.Iterator	4.90
java.beans.PropertyChangeListener	3.29
java.security.PrivilegedExceptionAction	3.18
java.util.Enumeration	1.55
javax.swing.event.ChangeListener	1.45
java.awt.event.ItemListener	1.33
java.util.Map$Entry	1.27
javax.swing.event.DocumentListener	1.24
javax.swing.event.ListSelectionListener	1.20
javax.swing.plaf.UIResource	1.13
java.lang.Comparable	1.08
java.util.ListIterator	0.79
java.io.FilenameFilter	0.75
javax.swing.Icon	0.66
java.lang.Cloneable	0.64
Total	**84.37**

Is there any set of super classes and interfaces in the Java standard library that is being used frequently to define inner classes?

We observed that only a few interfaces are implemented and super classes are extended repeatedly (cf. Table 6.13 and Table 6.14) from the Java standard library.[5] As expected, this finding suggests that the usage of the inner classes is aligned with the implementation of some specific common behaviors. We found that most of the inner classes are used significantly for Graphical User Interface (GUI) purposes. This also accounts for the observed interface and superclass frequencies. For example, both the `ActionListener` interface and the `AbstractAction` class that provide us with a useful mechanism to implement listeners for handling action events are found to be used with high frequencies (16.5% and 9%, respectively).

Based on the data set investigated, we observed that interfaces are being implemented more than extension of super classes while defining inner

[5] We used the Oracle's API specification [17] to identify the interfaces and super classes that belong to the Java standard library. We excluded, however, the default parent `java.lang.Object` in this case.

Table 6.14: 20 Most Frequent Super Classes (Library) extended by the Inner Classes, (These are 3% of the total super classes but contribute 60% of the total extensions)

Super Class	**Count (%)**
javax.swing.AbstractAction	9.01
java.lang.Enum	8.31
java.lang.Thread	7.72
java.util.AbstractSet	4.50
java.awt.event.MouseAdapter	4.43
java.awt.event.WindowAdapter	4.18
javax.swing.JPanel	2.35
java.lang.Error	1.95
java.util.AbstractCollection	1.83
java.awt.event.FocusAdapter	1.79
java.awt.event.KeyAdapter	1.79
java.lang.ThreadLocal	1.61
java.lang.RuntimeException	1.50
org.xml.sax.helpers.DefaultHandler	1.41
java.lang.Exception	1.40
javax.swing.border.AbstractBorder	1.27
javax.swing.DefaultListCellRenderer	1.27
java.io.InputStream	1.27
javax.swing.filechooser.FileFilter	1.18
java.lang.ref.WeakReference	1.09
Total	**59.69**

classes. Only 439 distinct interfaces are implemented 41,447 times. On the other hand, 841 super classes (excluding object) are extended 19,186 times. Such higher frequency of interface implementation indicates that developers tend to follow a much advised design guideline “favor object composition over class inheritance” [85]. For example, while implementing concurrent behaviors, we observed that developers implement the `java.lang.Runnable` interface (21.18%) more than they extend the `java.lang.Thread` class (7.72%).

While the above findings provide us with an insight into the parents of inner classes that belong the Java standard library, it would be useful to know about the parents that do not belong to that library. But the custom interfaces and super classes are diverse in nature and usually originate from a variety of sources (e.g., user defined, third party utility). Therefore, we do not provide a common list of such interfaces and super classes.

Instead, we present the proportion of parents of inner classes that belong to the Java standard library and that do not. Such comparative

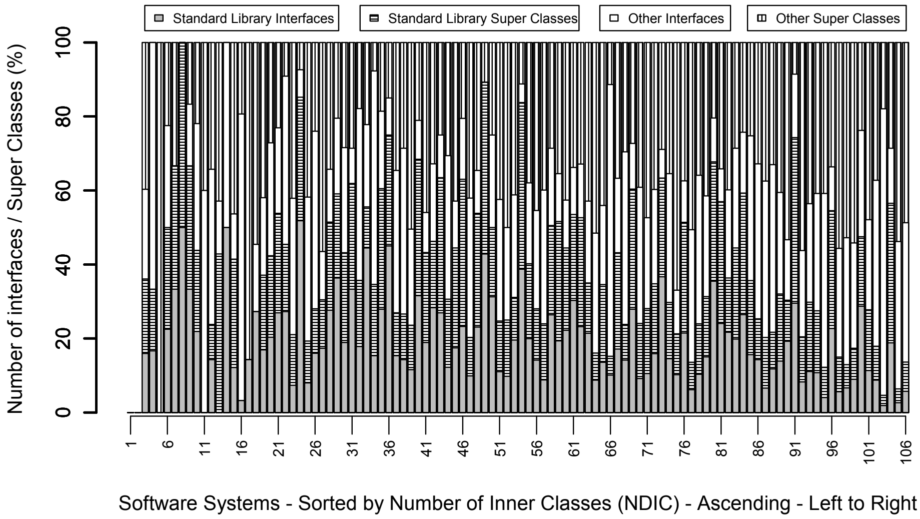


Figure 6.14: Number of unique interfaces implemented and super classes extended by inner classes of the software systems in the Qualitas Corpus.

proportions allow us to gain an insight into the extent of library and non-library types usage in defining inner classes in the software systems of the Qualitas Corpus.

We observed that developers, while defining inner classes, use Java standard library types with a varying degree of proportion (cf. Figure 6.14). Inner classes in almost all the software systems in the Qualitas Corpus are defined using the library types. Only two software systems (i.e., displaytag and javacc) do not make use of them. These software systems employ less than 15 inner classes only. On the other hand, the inner classes of only four software systems (i.e., xmojo, jsxe, jmoney, quickserver) use more than 80% library interfaces and super classes (LISC). These software systems employ less than 200 inner classes, and therefore may not represent the typical profile.

The median proportion of unique interfaces and super classes that belong to the Java standard library and also used in defining inner classes (LISC) is 35%. This suggests that 50% of the total software systems in the Qualitas Corpus make a substantial use of custom types (e.g., user defined, third party utility) as parents of inner classes. Inner classes in large software systems like eclipse and netbeans also employ a substantial number of custom types. Such custom types are often system-specific in nature. For example, 84% of the non-library interfaces and super classes (OISC) in eclipse belong to the package `org.eclipse`, suggesting system specific functionality implemented by the associated inner classes.

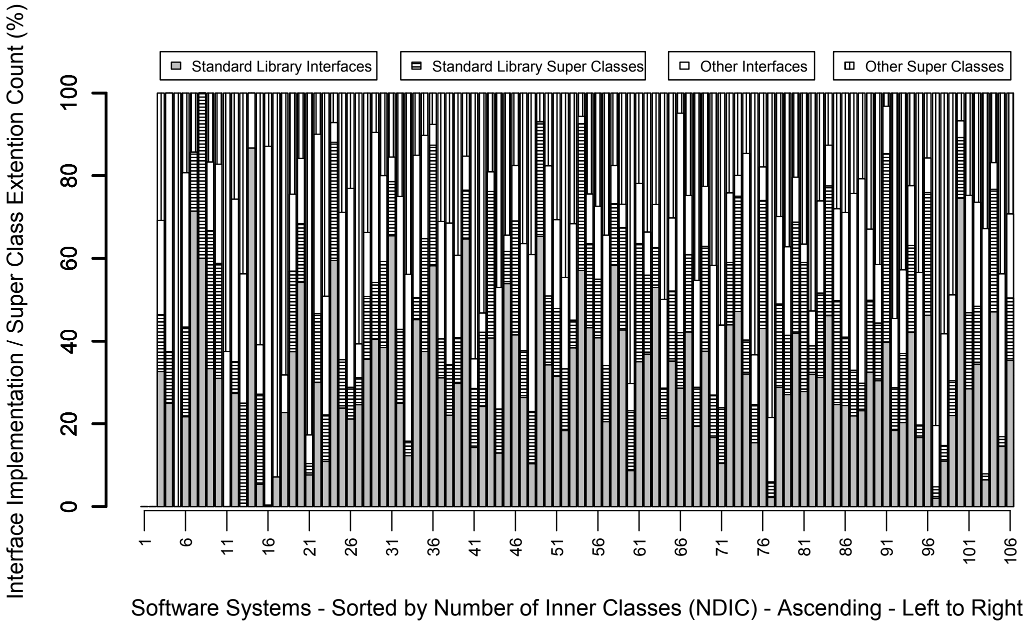


Figure 6.15: Number of times different interfaces and super classes are implemented and extended, respectively, by inner classes of the software systems in the Qualitas Corpus.

However, one common library interface or super class may be used as the parent of several inner classes. To gain an insight into the typical frequency of parents of inner classes, we investigated how many times a given interface or super class is implemented or extended, respectively, by an inner class. Figure 6.15 shows the proportion of frequency of library interfaces, library super classes, other interfaces, and other super classes in all the software systems of the Qualitas Corpus.

The median proportion of total number of implementations of library interfaces and extensions of library super classes (IELISC) is 47%. This proportion (IELISC) is more than 80% in 9 software systems (i.e., ireport, ivatagroupware, jtopen, joggplayer, jsXe, xmojo, jmoney, quickserver, and webmail). But most of these software systems do not comprise many inner classes (NDIC). In fact, all of these software systems, except ireport and jtopen, employ less than 200 inner classes. The software systems ireport (NDIC=1,746) and jtopen (NDIC=683) extensively rely on Java standard library while defining inner classes. In ireport, only 23 library interfaces are implemented 1301 times, and 15 library super classes are extended 256 times. On the other hand, only 23 non-library interfaces are implemented 70 times, and 19 non-library super classes are extended 118 time. This suggests that the library types are more frequently used in defining inner classes in ireport. We observed similar results in case of jtopen, too.

On the other hand, there are software systems in the Qualitas Corpus where the non-library interfaces and super classes (OISC) are used more frequently. In only 11% of the software systems, the proportion of non-library interface implementations and super class extensions (IEOISC) is above 80%. But most of these software systems make limited use of inner classes (less than 100). The software system azureus (NDIC=4358) is one of the exceptions. A substantial proportion of non-library interfaces and super classes (OISC) belong to its core packages. These are used as parents of inner classes repeatedly, causing the proportion of non-library interface implementations and super class extensions (IEOISC) to be higher than the library ones.

However, whether an inner class is defined using library types or user defined types in a particular software system depends on associated requirements, and therefore can vary across different software systems of the Qualitas Corpus. It is evident that the types in Java standard library does not serve as parents of all inner classes (as almost all the software systems comprise custom types as parents of inner classes). This causes developers to write their own types (or use third party utility) to satisfy system specific requirements while defining inner classes in Java-based software development.

6.3.5 Degree of Nestedness

The nesting profile of different inner classes of the investigated Java based software systems is depicted in Figure 6.16. Though most of the inner classes are found to be nested at level 1, it can rise up to

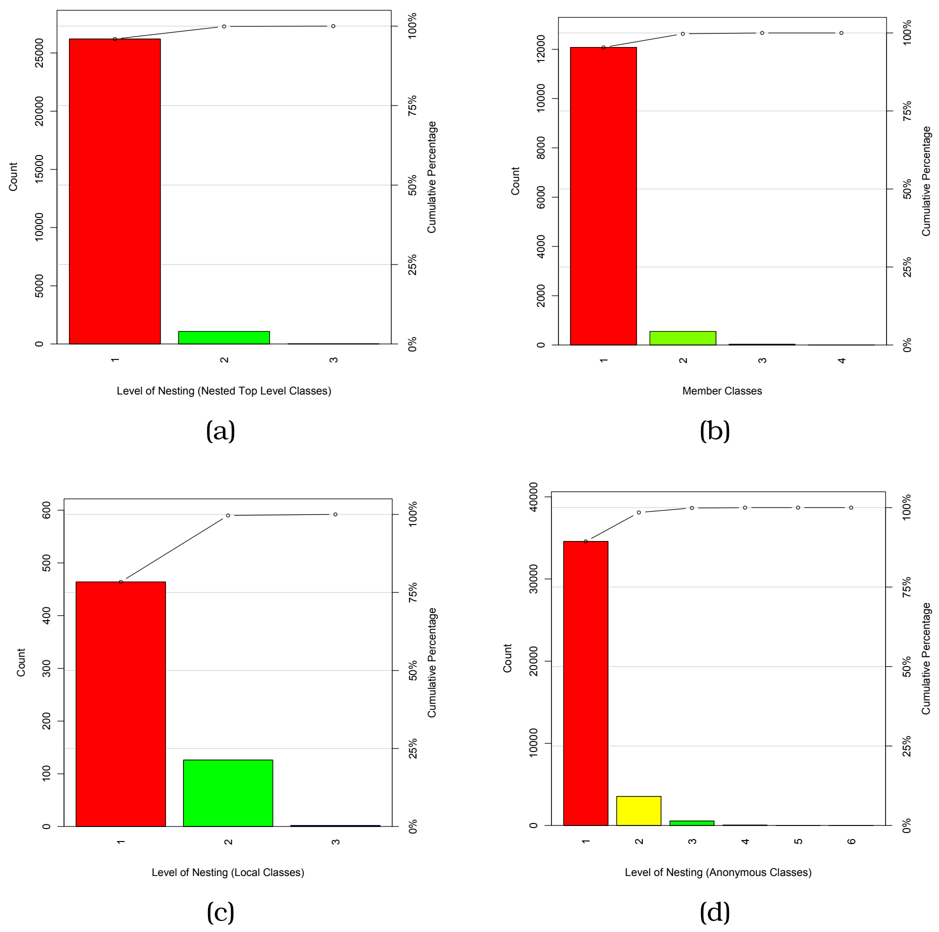
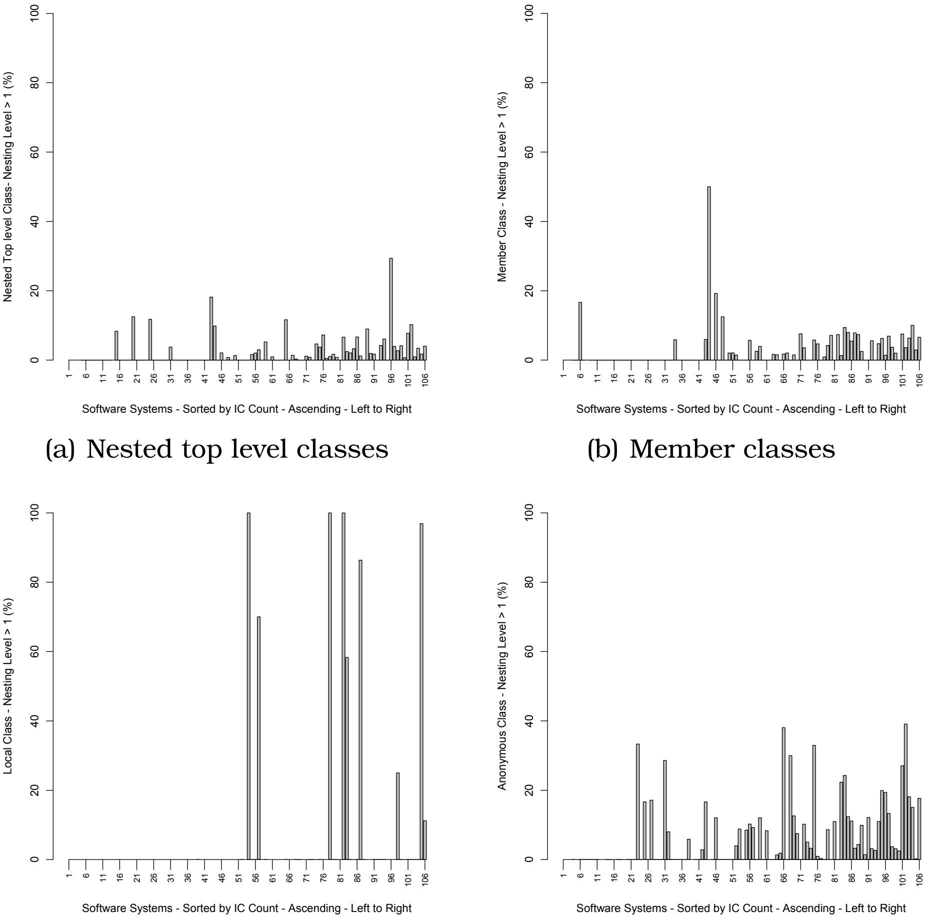


Figure 6.16: Level of nesting of various inner classes (a) Inner classes (b) Nested top level classes (c) Member classes (d) Anonymous classes

level 6. The highest nesting level of nested top level classes is 3 (27 classes), member classes is 4 (2 classes), local classes is 3 (2 classes), and anonymous classes is 6 (2 classes).

In 92% of the software systems in Qualitas Corpus, the proportion of nested top level classes that are nested at level 2 or more (NCNLTM) is less than 10% (cf. Figure 6.17(a)). The highest proportion (29.37%) of such NCNLTM is found in weka (NC=269, NCNLTM=79).

In 94% of the software systems, less than 10% of the member classes are nested at level 2 or more (cf. Figure 6.17(b)). The application emma employs two member classes, one nested at level 1 and the other at more than one, resulting in proportion of MCNLTM to be 50%.



(a) Nested top level classes

(b) Member classes

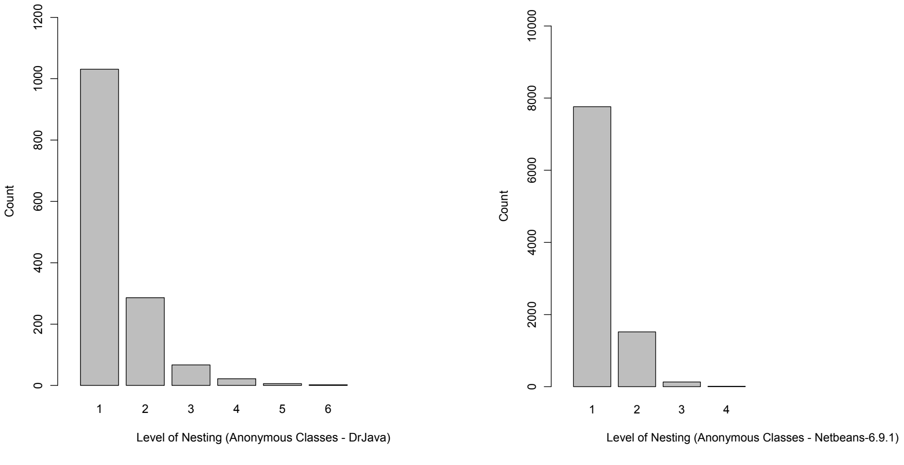
(c) Local classes

(d) Anonymous classes

Figure 6.17: Inner classes with nesting level more than one (Percentage is calculated with respect to corresponding classes. For example, percentage of nesting level >1 in anonymous classes of a given software system is computed based on total number of anonymous classes in that software system.)

We observed that a very limited number of software systems employ only a few local classes, and most of them are nested at level 2 or more. The proportion of LCNLTM appears high in Figure 6.17(c) due to only a few local classes. For example, some software systems that show more than 90% LCNLTM are xmojo (LC=12, LCNLTM=12), openjms (LC=1, LCNLTM=1), ganttproject (LC=3, LCNLTM=3), eclipse (LC=65, LCNLTM=63), aspectj (LC=22, LCNLTM=19).

About 66% of the software systems comprise less than 10% anonymous classes that are nested at level 2 or more (ACNLTM). Some software



(a)

(b)

Figure 6.18: Highest nesting level observed in anonymous classes in (a) DrJava, and (b) Netbeans

systems show above 30% ACNLTM. These are drjava (AC=1414, ACNLTM=383), jboss (AC=742, ACNLTM=290), trove (AC=255, ACNLTM=97).

The highest level of nesting in inner classes is found in *drjava* [11] - a lightweight development environment for Java. This software system provides an interaction interpreter (particularly for the beginners) that allows quick evaluation of Java expressions. For this purpose, drjava employs *dynamic java* - a Java source interpreter. In addition, it uses a standard *type checker* and a *type system* that allows for variance in different typing rules of the system.

The reason for such high level of nesting in drjava can be attributed to the system specific requirements (i.e., extensive type processing tasks required for providing interactive functionality for Java expression processing). In drjava, there are 2 anonymous classes at nesting level 6. They are defined in class *JLSTypeSystem* that deals with different type processing tasks (e.g., inferencing, subtyping) of the Java language specification. In addition, this class hosts 6 more anonymous classes at nesting level 5, and 22 more at nesting level 4. All of them are used for a similar purposes.

Apart from drjava, there are several software systems that comprise anonymous classes at nesting level 4. This includes azureus - a bittorrent client and netbeans - an integrated development environment for Java. The application azureus creates various listener (e.g., download manager listener, incoming message queue listener, disk manager read request listener) of its plugin interface by nesting anonymous inner classes repeatedly. Most of the highly nested anonymous classes in netbeans are found to be used for UI functionality.

Though the nesting level of various inner classes can vary across different software systems depending on the system specific requirements and the design choice of developers, it is evident (cf. Figure 6.16) that a large proportion of inner classes are defined at nesting level one. This suggests that developers do not tend to create highly nested program structures with inner classes and thus avoid complexity.

6.3.6 Breadth of Functional Decomposition

What is the typical distribution profile of methods in inner classes? Do developers maintain same profile of method distribution in both inner and outer classes? Do developers define inner classes with substantial number of methods?

Decomposition of Functionality - Overview

To gain an overview of the typical method distribution profiles in different inner classes, we computed the Gini coefficient of number of methods hosted by those classes. We also computed the Gini coefficient of methods in outer classes to yield a comparative understanding of method distribution in all types of classes. The resulting values are summarized in terms of box plot (cf. Figure 6.19). The interquartile range of method distribution in different inner classes is presented in Table 6.15.

We observed that the Gini coefficients of methods in all classes (NOMALLC) are high (with a median value of 0.64). The method distribution in outer classes (NOMOC) also exhibit almost same pattern (as represented by the corresponding Gini coefficients). In Qualitas Corpus, the median values of Gini coefficient of the methods in different classes exhibit a relation like *NOMALLC* > *NOMOC* > *NOMNC* > *NOMMC* > *NOMAC*

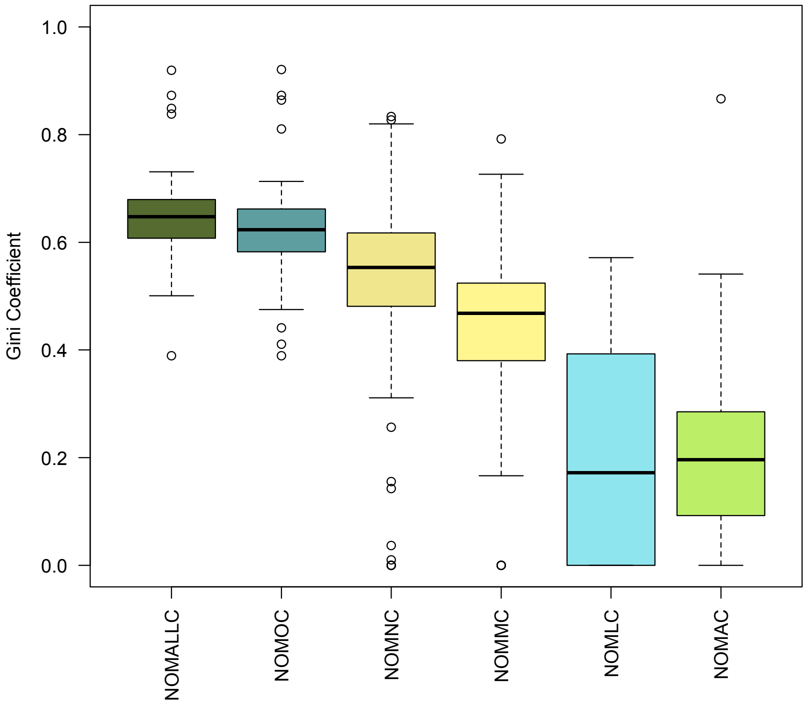


Figure 6.19: Boxplot of Gini coefficient of number of methods in different classes of the software systems in the Qualitas Corpus.

Table 6.15: The interquartile range (IQR) of the Gini coefficient (GC) of method distributions in different inner classes

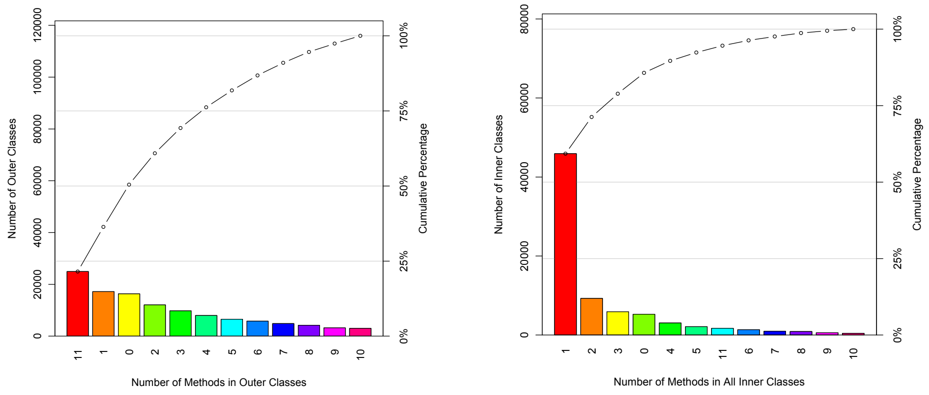
The interquartile range (IQR) of method distributions (NOM) in inner Classes		
Class Category	**GC (Q_3 - Q_1)**	**Interquartile Range (GC)**
All Classes	0.68 - 0.60	0.08
Outer Classes	0.66 - 0.58	0.08
Nested Top Level Classes	0.62 - 0.48	0.14
Member Classes	0.52 - 0.38	0.14
Local Classes	0.39 - 0.00	0.39
Anonymous Classes	0.28 - 0.09	0.19

(cf. Figure 6.19). As only few software systems employ local classes, we consider method distribution in local classes is not representative enough to make any conclusion.

As nested top level classes are just like top level classes but nested, they may comprise varying degree of methods - resulting in a distribution close to outer classes. This indicates that developers use top level and nested top level classes in a similar fashion (in terms of functionality decomposition). The least Gini coefficients of methods in anonymous classes further indicates that anonymous classes employ almost equal number of methods.

As inner classes facilitate functionality implementation of their respective enclosing classes (i.e., act as helper classes to address particular issues), we may expect a more equitable distribution pattern of methods in inner classes when compared to the distribution pattern of outer classes. As shown in Figure 6.19, the Gini coefficient of different inner classes confirms this.

However, there are some software systems in the Qualitas Corpus that comprise small number of inner classes (less than 20), and therefore result in a very high (above 0.85) or a very low (below 0.2) value of Gini coefficient of methods. These software systems include jasml, javacc, jparse, and cobertura. These software systems can be considered as exceptions.



(a)

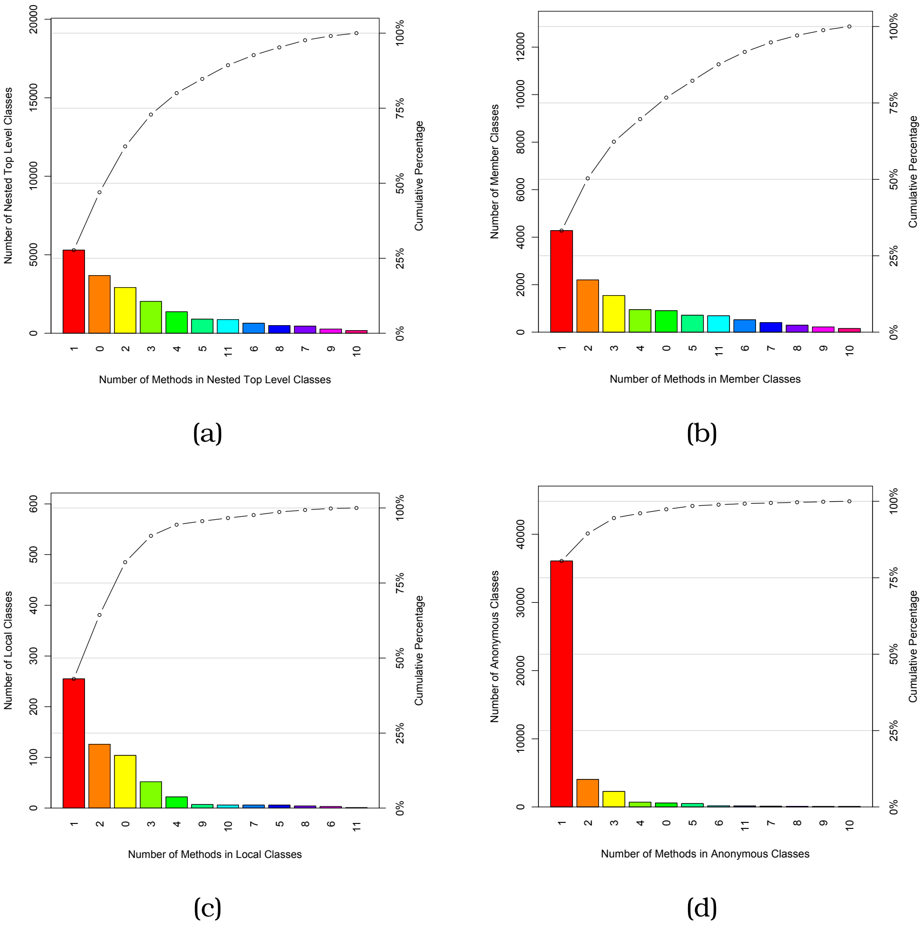
(b)

Figure 6.20: Number of methods* in different classes of the Qualitas Corpus (a) Outer classes, (b) Inner classes. (*Number of methods = 11 means more than 10 methods. X axis represents number of methods in a given class and Y axis represents corresponding number of classes. X axis is sorted based on the proportion of methods - higher to lower.)

Typical Number of Methods in Different Inner Classes

The above inequality-based analysis of methods informed us the nature of method distribution in inner classes. We do not know yet how many methods are typically employed in inner classes. For this reason, we counted number of methods in different classes of the software systems of the Qualitas Corpus. An overview of number of methods in inner and outer classes is presented in Figure 6.20.

We observed that developers define outer classes with a varying degree of proportion. About 78% of the outer classes comprise methods ranging from 0 to 10. The rest of the classes (22%) define more than 10 methods. On the other hand, developers define a substantial proportion (59%) of inner classes with a single method only.



(a) (b)

(c) (d)

Figure 6.21: Method count in different inner classes* (a) Nested top level classes (b) Member classes (c) Local classes (d) Anonymous classes (* Method count = 11 means more than 10 methods. X axis represents number of methods in a given class and Y axis represents corresponding number of classes. X axis is sorted based on the proportion of methods - higher to lower.)

The number of methods (NOM) can be used as an indicator of size (though NOM may not represent the actual size). The few number of methods in a class therefore may imply that developers tend to limit the size of inner classes. But which specific type of inner classes comprise few number of methods? To answer this question, we investigated the number of methods of each different type of inner classes (i.e., nested top level, member, local, and anonymous classes).

Figure 6.21 shows that the most inner classes comprises only a few methods. All four types of inner classes comprise substantial proportion of inner classes that comprise only single methods. The proportion of anonymous classes that contain a single method follows the Pareto principle [159] as more than 80% anonymous classes comprise only one method and the rest contains more methods (cf. Figure 6.21(d)).

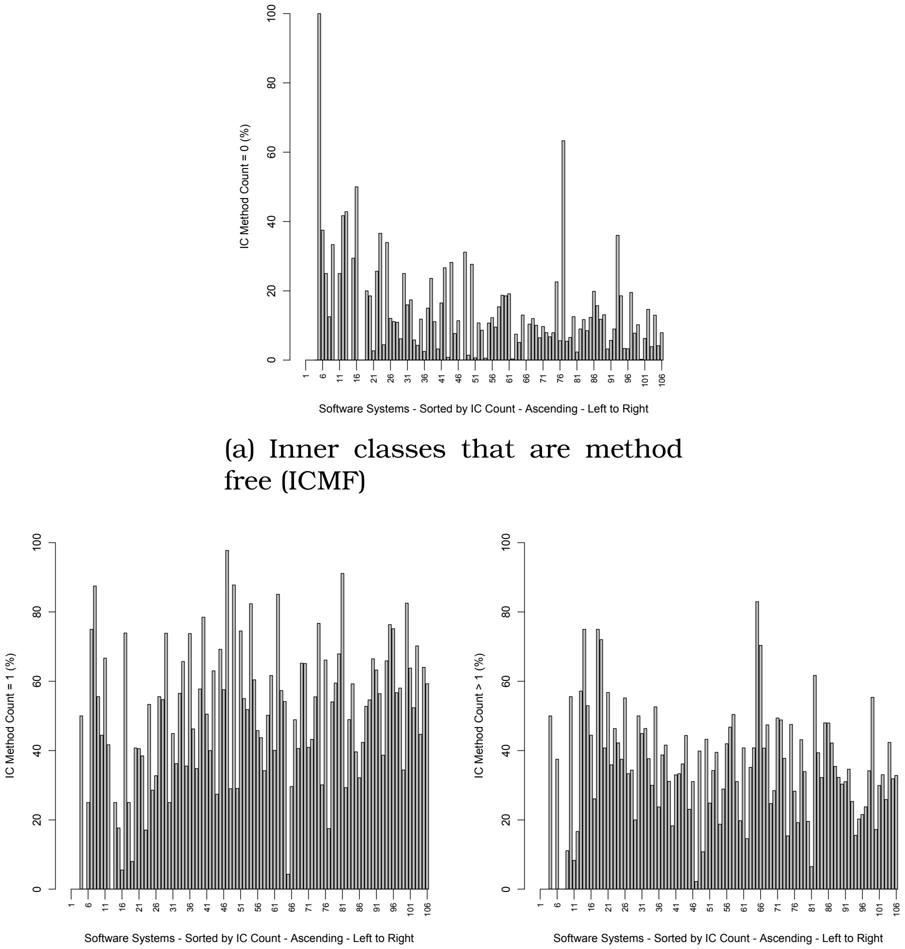
Apart from single method comprising inner classes, we observed that a significant proportion of inner classes do not contain any methods at all. What is the nature of such methods? To facilitate the analysis of method distribution in inner classes, we divide them into three categories based on the number of methods they contain:

- Inner Classes Without Any Methods
- Inner Classes With a Single Method
- Inner Classes With More Than One Method

Inner Classes Without Any Methods

Inner classes that do not define any methods at all usually contain *constructors* or *instance initializers* to initialize objects. We found in jruby, for example, a method free anonymous class that is used for environment configuration purpose (e.g., set current directory) through initialization of associated instance.

However, the use of method free inner classes in the software systems of the Qualitas Corpus is limited. The high proportion depicted in Figure 6.22(a) is due to few inner classes in the corresponding software systems. For example, jasml shows 100% as a result of few inner classes (7) in it. The median of the number of inner classes that do not contain any method is 18.



(a) Inner classes that are method free (ICMF)

(b) Inner Classes containing single method (ICSM)

(c) Inner classes with two or more methods (ICTMM)

Figure 6.22: Proportion of three types of Inner Classes: Method free, Only one method and More than one methods in Qualitas Corpus

On the other hand, two software systems (i.e., Fitlibraryforfitness and nakedobjects) comprise 333 and 256 (i.e., 63% and 36%, respectively) method free inner classes. These software systems, however, use *mock* objects supported by available mock object library (e.g., jmock[6]) and unit testing framework (e.g., junit[7]) for test automation tasks (i.e., test-driven development). The purpose of using inner classes without any method include supporting such task (e.g., creation of mock object).

[6] www.jmock.org
[7] www.junit.org

Inner Classes With a Single Method

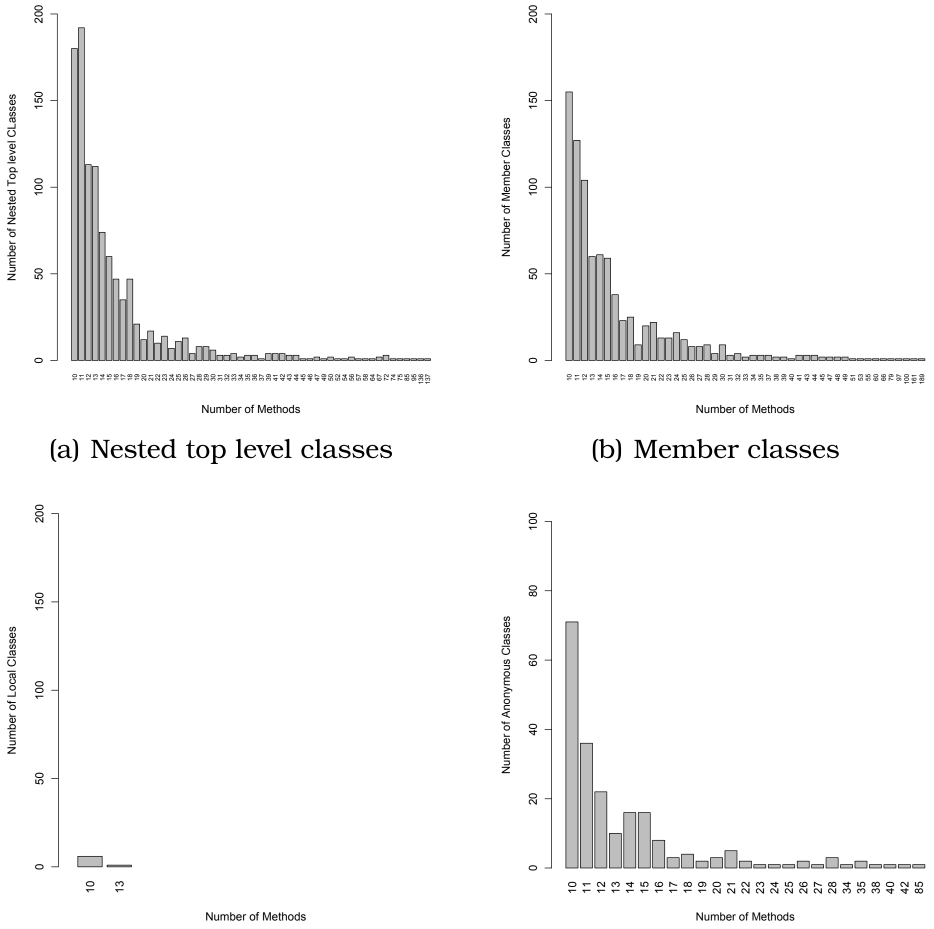
A single method in an inner class suggests one particular focus of that inner class. In Qualitas Corpus, a substantial proportion of inner classes (median 95) employ single methods only (cf. Figure 6.22(b)). Even in case of software systems that employ extensive inner classes (more than 1000, for example), the proportion of single method inner classes ranges from 45% to 82%. These include the two largest software systems, eclipse (13,985 inner classes) and netbeans (19,036 inner classes) that have 64% and 59% single method inner classes.

Inner Classes With More Than One Method

Apart from method free or single method inner classes, the third category includes inner classes that comprise two or more methods. As depicted in Figure 6.22(c), almost all the software systems employ a substantial proportion of inner classes that comprise two or more methods. This indicates that the size of the host inner classes are somewhat large (though NOM may not represent the actual size). Do developers employ inner classes with extensively high number of methods?

We found that there are inner classes that comprise a substantial number of methods. Figure 6.23 depicts the distribution of inner classes that have at least 10 methods. The maximum number of methods in different inner classes is quite high (NC=137, MC=189, LC=13, AC=85). Though such method-rich inner classes are not that frequent in their occurrence (i.e., less than 200), it would be intriguing to know why developers define such inner classes.

We found jre to employ a nested top level class that contains 137 methods. The class is concerned with graphics painting tasks (e.g., painting background of menubars and labels). The application cayenne contains a member class that employs 189 methods. The class is a final wrapper class (that includes mostly getters, predicates, etc.) and concerned with database functionality. Netbeans employ a local class with 13 methods. The application azureus employ an anonymous class that contains 85 methods. Most of the methods are getters, setter, and predicates.



(a) Nested top level classes

(b) Member classes

(c) Local classes

(d) Anonymous classes

Figure 6.23: Maximum number of methods found in different inner classes of the software systems in the Qualitas Corpus. X axis represents number of methods in a given inner class, and Y axis represents corresponding number of classes.

Table 6.16: The maximum number of methods in inner classes in 5 largest inner class employing software systems (starting from top of list) of Qualitas Corpus

Software Systems	Max. NOM in NCs	Max. NOM in MCs	Max. NOM in LCs	Max. NOM in MCs
netbeans	95	97	13	40
eclipse	85	47	10	42
jre	137	60	3	17
azureus	72	27	10	85
jboss	44	51	2	20

Based on the investigated software systems, we conclude that developers structure functionality using inner classes by defining fewer number of methods (when compared to number of methods in outer classes). Though there are some inner classes with a substantially high number of methods (presented in Figure 6.23), these inner classes can be considered as exceptions.

6.4 Summary

In this chapter, we investigated how developers use the concept of inner classes in Java-based software systems. Our observations revealed some insights into the state of current software development practices using inner classes. We summarize the key observations below.

- Developers use the notion of inner classes with a varying degree (ranging from 0% to 80%) in the software systems of the Qualitas Corpus. But the variants of inner classes (i.e., nested top level, member, local, and anonymous) are not used evenly. We observed a high use of all of them except the local ones. The underlying reasons could be that developers do not consider local classes useful in solution design. Whatever the reason is, the rare use of the concept of local class indicates a limited utility in Java-based software development, which in turn may merit its removal (or deprecation) from the Java language.

- When developers structure solutions using inner classes, they confine them in a relatively few number of host classes. The Gini coefficient of inner classes in most of the software systems in Qualitas Corpus is above 0.90, which indicates their highly uneven distribution profile. Such distribution is also evident across different domains (e.g., database, middleware), suggesting that the use of inner classes in Java-based software systems is domain-agnostic.

- Developers do not use inheritance much when defining inner classes. Yet, some software systems comprise inner classes that have relatively high value (up to 10) of DIT (depth in inheritance tree).

Such high DITs are often observed due to machine generated inner classes, and also implementation of GUI functionality. However, we found that only a few interfaces and super classes (of Java standard library [17]) are being repeatedly implemented and extended, respectively.

- Most of the inner classes are nested at level one. In only a few instances (less than 5 classes in entire Qualitas Corpus), we observed the nesting level 4 and higher. This finding suggests that developers do not tend to write highly nested code, and thus show a tendency to avoid complexity. In other words, they tend to comply with the advices (e.g., [95]) to avoid deep nesting, and thus contribute to better code readability - a key factor in software maintenance [178].

- A substantial proportion of inner classes comprises single methods only. In case of anonymous classes, this proportion follows the Pareto principle [159]. This suggests that developers may not intentionally make code clumsy, rather it is mandated by framework (e.g., SAM types). Therefore, the proposed lambda expression that targets SAM types could help developers to write program code for same functionality more concisely. Thus, readability of resulting program code may be improved.

Chapter 7

Conclusions

Understanding developer behaviors and decisions in formulating solution designs using programming language features, and also their tendencies in adhering to associated recommendations has been an active area of research (e.g., [74, 77, 93]). But our knowledge in this context still remains somewhat sketchy. In this thesis, our objective was to enrich our current understanding on developer tendencies in adhering to available recommendations in particular, and their practices in using programming language features in general.

To achieve our objective, we studied developer behaviors in using a set of features (i.e., fields, properties, and inner classes) of the Java programming language. Based on the fact that developer behaviors are imprinted into the software systems they produce, we conducted an empirical study on the Qualitas Corpus - a collection 106 open source Java-based software systems. We collected necessary software metrics data and analyzed them with statistical techniques (e.g., frequency distribution analysis and inequality analysis). We built descriptive models of software metrics to gain an understanding of developer behaviors, described our observations, and summarized the lessons learnt.

The key observations of this work can be summarized as below:

> *While using programming language features, developers tend to adhere to coding conventions, design guidelines, and advices they are provided with. They enjoy the flexibility (in using language features) offered to them and exhibit certain statistical consistency in structuring solutions using language features. Though available features govern developer design choices, they may not use a programming concept unless it offers a significant value.*

In this chapter, we revisit the contributions of this work (as presented in chapter - 1) and discuss them in section 7.1. We then present, in section 7.2, the implications of the outcome of this work. Finally, we discuss in section 7.3 the possible future directions that this work can take.

7.1 Contributions and Discussion

Developers tend to adhere to recommendations

We showed that developers tend to follow advices, design guidelines, and coding conventions associated with the fields, properties, and inner classes of the Java programming language.

- **Fields**

  We confirmed that fields are mostly defined as private (presented in Chapter 4, Section 4.3.1). This indicates developer tendencies in adhering to the information hiding principle [162] and advices (e.g., *all data should be hidden within its class* [183], *don't expose state if you don't have to* [12]). Though there are exceptions, the extent of violations is minimal. When developers violate recommendations and expose states (either deliberately or accidentally) by defining fields with non-private visibility modifiers, they take advantages (as represented by external access) only in few cases.

The exposed field hosting classes and exposed fields accessing classes exhibit similar distribution patterns (presented in Chapter 4, Section 4.3.4) - an emergent property visible at the entire Qualitas Corpus level (rather than individual system level). This indicates that there exists some form of relation between exposing fields and accessing them.

- **Properties**

  In case of properties, contrary to conventional belief, we found that they are neither commonplace nor evil. There are many advices regarding the use of getter and setter methods in solution design, both in favor of (e.g., do not change object's state without going through its public interface [183]) and against them (e.g., they circumvent private visibility modifiers and thus expose an object's state [69, 105, 106, 114]). Given such advices, we observed that developers proactively employ them in order to satisfy specific domain requirements, not just to circumvent data encapsulation. We observed that developers do not always accompany private fields with getter and setter methods. This is evident by scattered distribution of, and weak correlation between, private fields and pure getter and setter methods, discussed in Chapter 5, Section 5.3.3, in most of the software systems in the Qualitas Corpus.

  The purpose of using of getter and setter methods in Java-based software development is not just fabrication of read and write access, respectively. We observed that when developers define such methods, they do not always merely manifest read/write access, they implement some additional functionality in those methods.[1] Such practice is observed more in case of storing fields as evident by more occurrence of real setter methods than real getter methods - discussed in Chapter 5, Section 5.3.2).

[1] Whether such practices (i.e., implementing additional functionality in getter and setter methods) are good or bad has not been investigated. But considering the maintenance issue, a method should do what it is supposed to do (e.g., method *setColor* should set the field *Color* only).

- **Inner Classes**

  Given advices regarding the use of inheritance, both in favor (due to code reuse facility) and against (due to complexity associated with deeper classes in the inheritance hierarchy [192] and resulting maintenance difficulty [131]), we observed that developers do not use inheritance much when defining inner classes (discussed in Chapter 6, Section 6.3.4). Though some software systems comprise inner classes that have relatively high value (up to 10) of DIT, such cases are very rare and often due to machine generated inner classes and implementation of GUI functionality. This finding suggests that developers tend to limit the use of inheritance while defining inner classes.

  Given the debate concerning the reduced code readability induced by anonymous inner classes due to their clumsiness and bulky syntax [184], we found that most of them comprise a single method only (with a profile following the Pareto principle [159] - presented in Chapter 6, Section 6.3.6). This suggests that developers may not intentionally make code clumsy, rather it is an artifact of the induced application or framework requirements (e.g., SAM types). Besides, developers tend to comply with the advices (e.g., [95]) to avoid deep nesting, thus show a tendency to avoid complexity, and contribute to better code structure - a key factor in software maintenance [178].

However, the violations of recommendations, particularly involving the use of fields and properties, raise questions why developers do not adhere to given guidelines perfectly. Our observations, however, do not confirm that developers intentionally violate given advices or the advices are not worth following. One reason of violation of advices could be the associated development constraints. It is likely, for example, that time pressure can enforce priority on getting things done (i.e., production of just functional code, rather than code that is both functional and perfectly complies with available recommendations). Further investigation is required to confirm such hypothesis.

Developers, while using language features, enjoy the flexibility offered to them

We demonstrated that developers employ Java's property mechanism, a feature supported by code convention, as they desire. As a result, a variety of patterns emerge. We identified a catalog of 10 distinct patterns (described in Chapter 5, Section 5.2.1). These patterns capture different definitions of properties developers employ in practice. For example, a getter method is classified into three categories: pure, real, and virtual getter methods. While the pure getter method implement only read access to associated instance field, real and virtual getter methods are involved in implementing additional functionality.

An aspect of concern, however, is that developers often define methods with get- and set-semantics, but are not designated as such. We call them getter-like and setter-like methods. In addition, developers also define predicates (e.g., isTrue()), but with non-boolean return types. Such practices confirm violations of coding conventions and also indicates that violation may occur due to flexibility. However, the variety in employing the property mechanism indicates that developers utilize, according to their intentions, the flexibility of defining properties offered by the Java programming language (unlike built-in support for properties available in other languages like C#).

Developers exhibit certain consistency in using language features

We showed that there exists a *certain statistical consistency* (as represented by a small and narrow bounded region of computed Gini coefficients) among the developers in employing language features. The distribution profiles of the studied language features are highly concentrated in Java-based software systems, suggesting a consistent practice. We identified typical ranges (in terms of computed Gini coefficients and corresponding Lorenz curves of the distribution of studied features - presented in Chapter 4, Section 4.3.2; Chapter 5, Section 5.3; Chapter 6, Section 6.3.3) within which developers organize solution design.[2]

[2] This observation provides support for the notion of "decision frame", formulated by Tversky and Kahneman [210], that states that decision makers (e.g., developers)

The recorded IQR of Gini coefficients of different aspects of studied features (e.g., fields with various visibility modifiers, getter and setter methods, and various inner classes) can serve as reference to indicate accepted practice in organizing solutions using the associated language features. Though any deviation from such observed region does not necessarily imply a problem, its boundededness may be an indicator of some form of cognitive preferences of developers. In addition, any unusual Gini coefficient computed for a given feature in a software system can signal its (structural) difference from typical Java-based software systems.

However, even though developers concentrate the studied language features in a relatively few number of classes in general, we noticed that the concentration profiles of some of those features (e.g., getter and setter methods) are often negatively related to their proportions profiles. That is, the more features are employed in a software system, the more they are dispersed across the software system. This suggests that there is a tendency in developers to work with small and manageable classes, causing the distribution of features to be dispersed across more classes. This indicates that some sort of God-like aversive design strategy is practiced by developers.

Developers avoid some language features

We presented evidence (in Chapter 6, Section 6.3.1) that the concept of local classes are being used very rarely in the software systems of the Qualitas Corpus. Such rare use suggests that this concept is not well-accepted by developers, and also indicates that developers may not use a programming concept unless it offers significant value. The rare use of local classes, however, merits an exclusion of this concept from the Java programming language.

proactively organize solution design within well-established boundaries defined by cultural environment and personal preferences.

Support for language feature modification proposals

This work presents empirical evidence to support potential change in Java language features. We demonstrated that developers make use of SAM types substantially. If similar functionality could be implemented with more concise yet effective language constructs, developer burden could be substantially scaled down. As SAM types are one central aspect of lambda expressions [20], the proposals (e.g., [182]) suggesting anonymous classes to be replaced with lambda expression would benefit developers to write more concise and readable program code.

Classification of the software systems in the Qualitas Corpus

We classified the software systems in the Qualitas Corpus. We identified 12 major domains (e.g., middleware, database, games), based on the nature of functionality provided by these software systems. For example, the software systems apache derby and hsqldb are similar in nature as they offer data management functionality (e.g., data access, update, persistence). We grouped such software systems in the database domain.

The classification, presented in Chapter 3, Section 3.2.2, offers several benefits. For example, the classification can help maintaining diversity of the software systems in the Qualitas Corpus, and can also assist studies that involve domain-specific characterization of developer behaviors in using programming language features in Java-based software systems.

However, we do not claim that this classification is perfect. The primary purpose for this classification was to serve as a vehicle for investigating developer practices regarding the use of programming language features. Other interpretations are possible, and they may give rise to a refinement of this classification.

Support for study of programming language features

We contributed to the research community an extensible framework for metrics extraction and processing: *jCT - a Java Code Tomograph* [139]. It can be used to assist studies of developer behaviors in using programming language features, and also studies that involve software metrics data. Though jCT is built for Java, a similar approach (see Appendix B) can be adopted to study other languages (e.g., C#). In addition, jCT is equipped with different inequality measures (e.g., the Gini coefficients and Lorenz curve), as recommended by Vasa [216], to support inequality-based software analysis.

7.2 Implications

In this section, we present some of the implications of this work. These are (i) Impact on programming language design, (ii) Impact on education, (iii) Understanding developer behaviors, (iv) Recorded Gini coefficients as reference to the nature of language features distribution in typical Java-based software systems, (v) Supporting inequality-based software analysis, and (vi) Supporting software quality assurance.

Impact on Programming Language Design

A programming language has an impact on the developer productivity and maintenance. According to Scott [187], *"programming language feature has a huge impact on programmer's ability to write clear, concise, maintainable code, especially for very large systems"*. Consider an example of two programming languages, C++ and Java. Though both of them are object-oriented in nature, they offer different sets of language features, and therefore contribute in different ways to developers effectiveness. For example, investigating the productivity rate of programmers, a study [170] found that using Java in solution design yields significantly higher productivity rate than C++. This indicates that it is the set of language features that determines developers capability.

To remain effective, a language must evolve and be equipped with features that reflect developers intentions. According to Shaw [190], *"programming languages and methodologies evolve in response to the perceived needs of software designers and implementors"*. Due to the lacks of proper mapping of developers intentions, the resulting codes often become *clunky, bloated, or smelly* [91].

Language designers spend much of their time thinking about features [91]. The outcome of this work may assist them to think about both studied features and designing emerging features. For example, knowing that developers substantially use SAM types and rarely use the concept of local classes, language designers may consider to spend much attention to introduce more simple, yet effective language constructs for SAM types, on the other hand, they can decide to ignore (or deprecate) concepts like local classes.

Impact on Education

Software engineering education aims at delivering knowledge on tools and techniques for building software systems. For this purpose, software engineering courses include content on various aspects of software development life cycle (e.g., systems analysis, solution design, and implementation using programming language) that assist students (often novice) to become adept in software development practices. But it is found that turning novices into experts is a non-trivial task due to the given (limited) time frame [230].

Moreover, there is a persisting gap between software engineering education in academia and software development practices in industry. Both have some common and varying attributes. Mills [150] identified some commonality and variability of software engineering education both in industry and academia. To enrich software engineering education in academia, the involvement of industry practices could be useful.

For this purpose, educators in academia can use the observations of this thesis to enrich course content of software engineering modules to better reflect developers practices. In addition to the knowledge on programming language constructs (e.g., control structure, data structure, exception handling, and memory management), these modules can be decorated with knowledge on what developers actually do while formulating a solution using language features.

Thus, the outcome of this work may assist software engineering students to gain an insight into application of programming language features in contemporary Java-based software systems. For example, knowing that developers tend to follow experts advices may encourage novice students to adhere to advices, too. Moreover, knowing that private fields are not always accompanied by getter and setter methods may encourage students not to circumvent visibility modifiers through the property mechanism in Java. Another example could be shallow nesting structure of inner classes (i.e., most of the inner classes are not nested much) may discourage students to create highly nested program structure. Thus, software engineering students may become familiar with accepted practices in using language features as part of their education in academia.

Understanding Developer Behavior

Understanding the *human factor* in software development, particularly developer behaviors and their use of associated tools (e.g., programming languages) has been an important area of research. This has been demonstrated by significant related work (e.g., [36, 64, 86, 87, 127, 136, 193, 196, 226, 236]) that covers different human-centric aspects of software development and programming (e.g., cognitive issues in software development, psychological perspective of programming, and programming language features).

The outcomes of this work may assist researcher as it provides us with developer behavior in terms of structural organization of features in software systems that they built. Though developer behavior is not directly articulated through interaction with developers, this work presents their natural choices as manifested in, and inferred from, the product they built. More precisely, developers exhibit, while organizing solutions with language features, certain consistency (as represented by the narrow bounded regions of Gini coefficients). We attributed such consistency to some form of cognitive preferences of developers. As cognitive workload and comfort are related [36], such observed consistency may be considered as cognitive comfortability of developers, which implies that developers employ language features the way they are more comfortable with.

Researchers interested in investigating the psychological perspective of programming may benefited from the outcome of this work. They may correlate cognitive aspects of developers in association with some results (e.g., narrow bounded Gini coefficients) to uncover any potential relation between what developers think regarding solution design and what they actually do in practice. Moreover, observations like developers enjoy the flexibility in using language features may be useful to study their mental model from psychological perspective than possible today.

Supporting Inequality-based Software Analysis

Inequality-based software analysis is a recently emerged trend in studying and reasoning about contemporary software systems. Researchers employ inequality measures in their studies (e.g., [140, 216]) to understand different aspects (e.g., use of language feature) of software systems. But we have little tool support for such purpose, and therefore our capability in conducting empirical studies becomes somewhat restricted. According to Dijkstra [67], *"the tools we use have a profound (and devious) influence on our thinking habits, and, therefore on our thinking abilities"*.

The methodology followed, and framework (jCT) devised, in this work may assist researchers interested in similar type of study. We demonstrated how developers practices in using language features can be reasoned about through the Gini coefficient. In addition, unlike many other tools (e.g., JHawk [223], SonarJ [14], Mutations [9]), jCT offers support for inequality-based software analysis.

Supporting Software Quality Assurance

The coding standards and guidelines, if followed appropriately, can reduce overhead involving various issues associated with program code (e.g., maintainability, correctness, readability). Such guidelines are, in general, *suggestive*, and therefore developers may or may not comply with them. But considering the resulting benefits of complying with such guidelines, it is expected that developers should follow them.

We demonstrated that developers exhibit a tendency to follow the studied coding standards with minimal violations (similar to the competent programmer hypothesis [161] that states a competent programmer write code that is close to being correct). The findings of this work present empirical evidence to support it. For example, though developers define properties by adhering to the Java coding conventions [10], in limited cases they do not follow such conventions which is evident by the existence of *getter-like* and *setter-like* methods in the software systems of Qualitas Corpus.

Software managers, however, can utilize above observations (i.e., developers do not perfectly follow advices) to devise any potential corrective measures. For example, one measure could be initiating action either to ensure perfect adherence to coding guidelines or to improve quality assurance activities (e.g., code maintainability) based on the fact that some code, though functional, are not structured as expected. Existing tools and IDEs can be enriched with rules that monitor such cases. For example, when developers ignore conventions much, IDEs should automatically detect such cases and warn developers. This would be useful as higher proportions of violations (e.g., getter-like and setter-like methods) may be problematic considering the maintenance issues (difficult to quickly identify the purpose of such methods).

7.3 Limitations and Future Work

In this section, we describe the limitations of this work and some directions of future work. These are grouped into (i) Beyond Java, (ii) Incorporating developers feedback with language feature usage patterns, (iii) Relating software quality attributes with language features distribution patterns, (iv) Language features usage patterns in evolving software systems, (v) Model building, (vi) Refinement of the classification of properties, and (vii) Refinement of the classification of the software systems in Qualitas Corpus.

Beyond Java

Our primary focus in this thesis was to investigate the language features (particularly, fields, properties and inner classes) usage patterns of the Java programming language. We did not explore different programming languages for the same purpose. We consider this as one of the major limitations of this work as it resists us to portray a comparative scenario of similar features usage profile in different contexts (e.g., C#).

However, we speculate that a study on other object oriented programming language would result in similar observations. If not, the potential reasons include the semantic variations offered by different languages (e.g., C++, C#). It would be worthwhile to explore this research avenue further as it provides us with the opportunity to compare features usage patterns as practiced by the developers of different programming languages.

Moreover, a longitudinal study focusing on the comparative analysis of the underlying factors that influence feature usage patterns of a diverse set of programming languages can uncover the effectiveness of particular features. This can guide language designers to address the potential issues to equip the emerging languages with better and more focused language semantics.

Incorporating Developers Feedback with Language Feature Usage Patterns

To understand programming language feature usage patterns and developers preferences in employing different features while fabricating solution design, we completely relied on the open source software systems that the developers produce. We did not involve any software developers directly in our study to understand their viewpoints regarding the application of different programming language features during software development.

Therefore, it would be another interesting avenue of research to investigate what actually motivates them while employing different features of the adopted programming language. This can be accomplished through different means (e.g., survey, interview, questionnaires) that allow us to gain more direct feedback. In addition to the involvement of the developers only, it would be worth capturing both the technical and non-technical aspects of the entire development context. For example, the development environment, associated cultural, psychological and social factors (e.g., interaction with team members), and development pressure can be considered to yield a better insight into the driving factors of programming language feature usage patterns.

Relating Software Quality Attributes with Language Features Distribution Patterns

The investigations of the correlations between the programming language features usage patterns and software quality attributes (e.g., readability, maintainability, reliability, testability) is another potential avenue of research. Though these issues are addressed in different studies [42, 65, 98, 171], the impact of fields, properties and inner classes on software quality remain mostly unexplored. An extensive investigation addressing the quality aspects of these features can be useful. Moreover, a longitudinal study focusing on uncovering the associations of software quality attributes with extreme distributions patterns (i.e., highly even or highly uneven) of features can result in valuable feedback for both design and quality assurance activities in software engineering.

Language Features Usage Patterns in Evolving Software Systems

Investigation of the programming language features usage patterns in the evolving software systems is another potential avenue of research. It would be worth inspecting how the usage patterns of fields, properties and inner classes evolve in successive releases of software systems. Such study can reveal useful insights into the effect of language features in supporting software evolution. For example, the evolving patterns of data storage and retrieval mechanisms fabricated in the different releases of software systems can provide us with insights to monitor how developers manage data functionality throughout the lifecycle of software systems.

Model Building

There is a growing interest in Bayesian analysis (*e.g.*, [57,200,213,232]) of different aspects of quality of software systems. For example, Koten and Gray [213] developed a maintainability prediction model for object-oriented software systems. The construction of the model is based on software metrics data (e.g., described in [56, 131]) that captures different aspects of object-oriented concepts. Researchers also increasingly adopting Neural network based techniques [101, 113, 119, 199]. For example, Quah and Thwin [176] applied neural network on object-oriented metrics (including the CK metrics [56]) for software quality estimation. Such models can be built based on the fields, properties, and inner classes measures, and also correlation can be established between their Gini coefficients and quality attributes (e.g., reliability, maintainability).

Refinement of the Classification of Properties

We classified getter and setter methods into 12 different types that capture their different variants. This classification can be refined further by including various types of properties (e.g., bound properties, dynamic properties, vetoable properties). It would be interesting to see how often they are being used in Java-based software development.

Refinement of the Classification of the Software Systems in Qualitas Corpus

A potential avenue of further research would be refining the classification of the software systems in Qualitas Corpus. We classified them in 12 categories (e.g., IDE, SDK). In our classification, there are some domains (i.e., middleware and tools) that comprise comparatively more software systems than other domains (e.g., database). Such domains might be split into sub-domains.

Besides, it would be worthwhile to consider the orthogonal system characteristics of the constituent software systems in Qualitas Corpus and place some of them into multiple categories (e.g., pmd, currently classified as testing tool, has also significant characteristics that warrant a secondary classification: IDE).

References

[1] Software Product Quality - Part 1: Quality Model, 1998. Retrieved August 15th 2011 http://www.sqa.net/iso9126.html.

[2] JMT - Java Measurement Tool. http://www-ivs.cs.uni-magdeburg.de/sw-eng/agruppe/forschung/tools/jmt.html, 2002.

[3] CyVis - Software Complexity Visualizer. http://cyvis.sourceforge.net/, 2005.

[4] Java Coding Standards. http://software.ucv.ro/~eganea/SoftE/JavaCodingStandards.pdf, 2005.

[5] BCEL: The Byte Code Engineering Library. http://jakarta.apache.org/bcel, 2006.

[6] Dependency Finder. http://depfind.sourceforge.net, 2008.

[7] FindBugs^{TM} – Find Bugs in Java Programs. http://findbugs.sourceforge.net, 2009.

[8] JDepend Tool. http://www.clarkware.com/software/JDepend.html, 2009.

[9] Mutations. http://www.ict.swin.edu.au/research/projects/helix/mutations.html, 2010.

[10] Code Conventions for the Java Programming Language. http://www.oracle.com/technetwork/java/codeconv-138413.html, 2011.

[11] DrJava – A Programming Environment for Java. http://www.drjava.org/, 2011.

[12] Often Overlooked Object Oriented Programming Guidelines. http://www.perlmonks.org/?node_id=317520, 2011.

[13] Publications based on the Qualitas Corpus. `http://qualitascorpus.com/docs/publications.html`, Apr. 2011.

[14] SonarJ - Visualize and Understand. `http://www.hello2morrow.com/products/sonarj`, 2011.

[15] C# Programming Guide. `http://msdn.microsoft.com/en-us/library/ff926074.aspx`, 2012.

[16] Central Washington University, Java Programming Style Guide CS 110. `http://www.cwu.edu/~gellenbe/javastyle/`, 2012.

[17] Java Platform, Standard Edition 7 API Specification. `http://docs.oracle.com/javase/7/docs/api/`, 2012.

[18] State of Lambda. `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html`, 2012.

[19] The Java Tutorials. `http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html`, 2012.

[20] Translation of Lambda Expressions. `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html`, 2012.

[21] Properties. `http://docs.oracle.com/javase/tutorial/javabeans/writing/properties.html`, 2013.

[22] Ruby on Rails. `http://rubyonrails.org/`, 2013.

[23] J. S. Alghamdi, R. A. Rufai, and S. M. Khan. OOMeter: A Software Quality Assurance Tool. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 190–191, 2005.

[24] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 53–64, 2007.

[25] G. Arango. Domain analysis: from art form to engineering discipline. *SIGSOFT Softw. Eng. Notes*, 14(3):152–159, 1989.

[26] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, May 1996.

[27] A. Baddeley. Working memory. *Science*, 255(5044):556–559, 1992.

[28] R. Barker and E. Tempero. A Large Scale Empirical Comparison of Object Oriented Cohesion Metrics. In *Asia Pacific Software Engineering Conference (APSEC)*, pages 414–421, Nagoya, Japan, Dec. 2007.

[29] A. L. Baroni and O. B. E. Abreu. An OCL-based Formalization of the MOOSE Metric Suite. In *In Proceedings of ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering*, 2003.

[30] W. Basalaj. Correlation Between Coding Standards Compliance and Software Quality. *IEE Seminar Digests*, 2005(11311):46–46, 2005.

[31] V. Basili. The Role of Experimentation in Software Engineering: Past, Current, and Future. In *Software Engineering, 1996., Proceedings of the 18th International Conference on*, pages 442–449, Mar. 1996.

[32] V. Basili, L. Briand, and W. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[33] V. Basili, F. Shull, and F. Lanubile. Building Knowledge Through Families of Experiments. *Software Engineering, IEEE Transactions on*, 25(4):456 –473, Jul./Aug. 1999.

[34] V. R. Basili. The Experimental Paradigm in Software Engineering. In *Proceedings of the International Workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions*, pages 3–12, London, UK, 1993. Springer-Verlag.

[35] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the Shape of Java Software. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications*, volume 41 of *ACM SIGPLAN Notices*, pages 397–412, Portland, Oregon, 2006.

[36] N. Bevan and M. Azuma. Quality in Use: Incorporating Human Factors into the Software Engineering Lifecycle. In *Software Engineering Standards Symposium and Forum, 1997. 'Emerging International Standards'. ISESS 97., Third IEEE International*, pages 169–179, Jun. 1997.

[37] P. Bhattacharya and I. Neamtiu. Assessing Programming Language Impact on Development and Maintenance: A Study on C and C++. In *Proceedings of the 33rd International Conference on*

Software Engineering, ICSE'11, pages 171–180, New York, NY, USA, 2011. ACM.

[38] A. Bonaccorsi and C. Rossi. Why Open Source Software Can Succeed. *Research Policy*, 32(7):1243–1258, 2003.

[39] A. Bonaccorsi and C. Rossi. Comparing Motivations of Individual Programmers and Firms to Take Part in the Open Source Movement: From Community to Business. *Knowledge, Technology and Policy*, 18:40–64, 2006.

[40] L. Briand, K. El Emam, and S. Morasca. On the Application of Measurement Theory in Software Engineering. *Empirical Software Engineering*, 1(1):61–88, Jan. 1996.

[41] J.-P. Briot and R. Guerraoui. On the Use of Smalltalk for Concurrent and Distributed Programming. In *Special Issue on Smalltalk (Language, Tools and Environment)*, Informatik/Informatique, Swiss Informaticians Society, Switzerland, Feb. 1996.

[42] F. Brito e Abreu and W. Melo. Evaluating the Impact of Object-oriented Design on Software Quality. In *Proceedings of the 3rd International Software Metrics Symposium*, pages 90–99, Mar. 1996.

[43] A. Brooks, M. Roper, M. Wood, J. Daly, and J. Miller. Replication's Role in Software Engineering. In F. Shull, J. Singer, and D. I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 365–379. Springer, Heidelberg, 2008.

[44] K. B. Bruce, A. Schuett, R. van Gent, and A. Fiech. PolyTOIL: A Type-safe Polymorphic Object-Oriented Language. *ACM Trans. Program. Lang. Syst.*, 25(2):225–290, Mar. 2003.

[45] B. Bruegge and A. H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java Second Edition*. Prentice-Hall, 2004.

[46] E. Bruneton. ASM 3.0: A Java Bytecode Engineering Library. `http://download.forge.objectweb.org/asm/asm-guide.pdf`, 2007.

[47] A. Buckley. JSR 202: Java(TM) Class File Specification Update, Dec. 2006. `http://jcp.org/en/jsr/detail?id=202`.

[48] F. Bulback. *Programming Delphi Custom Components*. M&T Books, 1996.

[49] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger. How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 23–32, Waikiki, Honolulu, HI, USA, 2011.

[50] M. Cartwright and M. Shepperd. An Empirical Investigation of an Object-Oriented Software System. *IEEE Transactions on Software Engineering*, 26(8):786–796, 2000.

[51] C. Chambers. The Cecil Language, Specification and Rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington 98195 USA, 1993.

[52] C. Chambers. The Diesel Language Specification and Rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington 98195 USA, 2006.

[53] S. Chiba. Load-Time Structural Reflection in Java. In E. Bertino, editor, *Proceedings ECOOP 2000*, LNCS 1850, pages 313–336, Cannes, France, Jun. 2000.

[54] S. Chidamber, D. Darcy, and C. Kemerer. Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis. *IEEE Transactions on Software Engineering*, 24(8):629–639, Aug 1998.

[55] S. R. Chidamber and C. F. Kemerer. Towards a Metrics Suite for Object Oriented Design. In *Proceedings of Conference on Object-oriented programming systems, languages, and applications*, OOPSLA '91, pages 197–211, Phoenix, Arizona, USA, 1991.

[56] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, Jun. 1994.

[57] S. Chulani, B. Boehm, and B. Steece. Bayesian Analysis of Empirical Software Engineering Cost Models. *IEEE Transactions on Software Engineering*, 25(4):573–583, Jul. 1999.

[58] Classycle Dependency Analyzer, Classycle Project. `http://classycle.sourceforge.net/`, 2010.

[59] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings OOPSLA 2000*, volume 35 of *ACM SIGPLAN Notices*, pages 130–146, Oct. 2000.

[60] S. Cohen and L. M. Northrop. Object-Oriented Technology and Domain Analysis. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 86, Washington, DC, USA, 1998.

[61] G. Concas, M. Marchesi, S. Pinna, and N. Serra. Power-Laws in a Large Object-Oriented Software System. *IEEE Transactions on Software Engineering*, pages 687–708, 2007.

[62] S. Counsell and P. Newson. Use of Friends in C++ Software: An Empirical Investigation. *Journal of Systems and Software*, 53(1):15–21, 2000.

[63] N. Cowan. The Magical Number 4 in Short-term Memory: A Reconsideration of Mental Storage Capacity. *The Behavioral and Brain Sciences*, 24:87–114, 2001.

[64] B. Curtis, I. Forman, R. Brooks, E. Soloway, and K. Ehrlich. Psychological perspectives for software science. *Information Processing & Management*, 20(1-2):81–96, 1984.

[65] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study. In *Software Maintenance, 1995. Proceedings., International Conference on*, pages 20–29, oct 1995.

[66] T. DeMarco. *Controlling Software Projects: Management, Measurement and Estimation*. Yourdon Press New York, 1982.

[67] E. W. Dijkstra. How Do We Tell Truths That Might Hurt? *SIGPLAN Not.*, 17:13–15, May 1982.

[68] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting Empirical Methods for Software Engineering Research. In F. Shull, J. Singer, and D. I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer London, 2008.

[69] P. J. Eby. Python Is Not Java. `http://dirtsimple.org/2004/12/python-is-not-java.html`, 2004.

[70] M. El Wakil, A. El Bastawissi, M. Boshra, and A. Fahmy. A Novel Approach to Formalize and Collect Object-Oriented Design-Metrics. In *Proceedings of the 9th International Conference on Empirical Assessment in Software Engineering*, 2005.

[71] R. Englander. *Developing Java Beans*. O'Reilly, 1997.

[72] M. English, J. Buckley, and T. Cahill. A Replicated and Refined Empirical Study of the Use of Friends in C++ Software. *J. Syst. Softw.*, 83(11):2275–2286, Nov. 2010.

[73] M. English, J. Buckley, T. Cahill, and K. Lynch. An Empirical Study of the Use of Friends in C++ Software. In *Proceedings of the 13th International Workshop on Program Comprehension*, IWPC '05, pages 329–332, 2005.

[74] M. English and P. McCreanor. Exploring the Differing Usages of Programming Language Features in Systems Developed in C++ and Java. `http://www.ppig.org/papers/21st-english.pdf`, 2010.

[75] J. Estublier, G. Vega, P. Lalanda, and T. Leveque. Domain Specific Engineering Environments. In *Proc. 15th Asia-Pacific Software Engineering Conf. APSEC '08*, pages 553–560, 2008.

[76] E. Evans. *Domain-Driven Design – Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[77] T. Ewan, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Proceedings of 17th Asia Pacific Software Engineering Conference*, pages 336–345, Sydney, Australia, Dec. 2010.

[78] X. Fang. Using a Coding Standard to Improve Program Quality. In *Proceedings of Second Asia-Pacific Conference on Quality Software*, pages 73–78, Dec. 2001.

[79] N. Fenton. Software Measurement: A Necessary Scientific Basis. *IEEE Trans. Softw. Eng.*, 20:199–206, Mar. 1994.

[80] N. Fenton, S. Pfleeger, and R. Glass. Science and Substance: A Challenge to Software Engineers. *IEEE Software*, 11(4):86–95, Jul. 1994.

[81] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.

[82] N. E. Fenton and M. Neil. A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

[83] D. Flanagan. *Java in a Nutshell*. OŔeilly and Associates, Inc., third edition, 1999.

[84] D. Flanagan. *Java in a Nutshell.* O'Reilly Media, fifth edition, Mar. 2005.

[85] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[86] J. Gannon. Human Factors in Software Engineering. *Computer*, 12:6–7, 1979.

[87] J. D. Gannon and J. J. Horning. The Impact of Language Design on the Production of Reliable Software. In *Proceedings of the international conference on Reliable software*, pages 10–22, 1975.

[88] J. Gil and I. Maman. Micro Patterns in Java. In *Proceedings OOPSLA 2005*, volume 40 of *ACM SIGPLAN Notices*, pages 97–116, San Diego, USA, Oct. 2005.

[89] C. Gini. Measurement of Inequality of Incomes. *The Economic Journal*, 31(121):124–126, Mar. 1921.

[90] M. Godfrey and Q. Tu. Evolution in Open Source Software: A Case Study. In *Proceedings of International Conference on Software Maintenance (ICSM 2000)*, pages 131–142, Los Alamitos CA, 2000.

[91] B. Goetz. Using Real-world Data to Drive Language Evolution Decisions. `http://public.dhe.ibm.com/software/dw/java/j-ldn1-pdf.pdf`, 2010.

[92] O. Goloshchapova. "Automated Inequality Analysis of Evolving Software Systems". Master's thesis, Swinburne University of Technology, Apr. 2013.

[93] T. Gorschek, E. Tempero, and L. Angelis. A Large-Scale Empirical Study of Practitioners' Use of Object-Oriented Concepts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE'10, pages 115–124, Cape Town, South Africa, 2010.

[94] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison-Wesley, 3rd edition, 2005.

[95] B. Guzel. Top 15 Best Practices for Writing Super Readable Code. `http://net.tutsplus.com/tutorials/html-css-techniques/top-15-best-practices-for-writing-super-readable-code`, 2012.

[96] S. Harker, K. Eason, and J. Dobson. The Change and Evolution of Requirements as a Challenge to the Practice of Software Engineering. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 266–272, Jan. 1993.

[97] T. Harmer and F. Wilkie. An Extensible Metrics Extraction Environment for Object-oriented Programming Languages. In *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 26–35, 2002.

[98] R. Harrison, S. Counsell, and R. Nithi. Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems. *University of Keele*, 52:173–179, 1999.

[99] B. Henderson-Sellers. Object-Oriented Metrics. In *TOOLS (11)*, page 548, 1993.

[100] G. Hertel, S. Niedner, and S. Herrmann. Motivation of Software Developers in Open Source Projects: An Internet-based Survey of Contributors to the Linux Kernel. *Research Policy*, 32(7):1159–1177, 2003.

[101] R. Hochman, T. Khoshgoftaar, E. Allen, and J. Hudepohl. Evolutionary neural networks: A robust approach to software reliability problems. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pages 13–26, 2-5 1997.

[102] H. Hofmann and F. Lehner. Requirements Engineering as a Success Factor in Software Projects. *Software, IEEE*, 18(4):58–66, Jul./Aug. 2001.

[103] A. Holkner and J. Harland. Evaluating the Dynamic Behaviour of Python Applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC '09, pages 19–28, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[104] N. M. Holtz and W. J. Rasdorf. An Eevaluation of Programming Languages and Language Features for Engineering Software Development. *Engineering with Computers*, 3:183–199, 1988.

[105] A. Holub. Why Getter and Setter Methods are Evil: Make Your Code More Maintainable by Avoiding Accessors. `http://www.javaworld.com/javaworld/jw-09-2003/jw-0905-toolbox.html?page=3`, 2003.

[106] A. Holub. More on Getters and Setters: Build User Interfaces Without Getters and Setters. `http://www.javaworld.com/`

javaworld/jw-01-2004/jw-0102-toolbox.html?page=1, 2004.

[107] T. Howles. Fostering the Growth of a Software Quality Culture. *SIGCSE Bull.*, 35(2):45–47, June 2003.

[108] W. S. Humphrey. The Software Engineering Process: Definition and Scope. *SIGSOFT Softw. Eng. Notes*, 14:82–83, April 1988.

[109] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[110] A. Hunt and D. Thomas. Tell, Don't Ask. http://pragprog.com/articles/tell-dont-ask, 2012.

[111] IEEE. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, page 1, 1990.

[112] IEEE Standard 1061-1998. *IEEE Standard for a Software Quality Metrics Methodology*, 1998.

[113] C. Jin, S.-W. Jin, J.-M. Ye, and Q.-G. Zhang. Quality Prediction Model of Object-oriented Software System using Computational Intelligence. In *2nd International Conference on Power Electronics and Intelligent Transportation System (PEITS)*, volume 2, pages 120–123, Dec. 2009.

[114] G. Jorgensen. Doing it wrong: getters and setters. http://typicalprogrammer.com/?p=23, 2008.

[115] JSeat. http://code.google.com/p/jseat, 2008.

[116] N. Juristo and A. Moreno. *Basics of Software Engineering Experimentation.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001.

[117] C. Kaner and W. P. Bond. Software Engineering Metrics: What Do They Measure and How Do We Know? In *Proceedings of the 10TH International Software Metrics Symposium - Metrics 2004.* IEEE, 2004.

[118] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Nov. 1990.

[119] S. Kanmani, V. R. Uthariaraj, V. Sankaranarayanan, and P. Thambidurai. Object-oriented software fault prediction using neural networks. *Information and Software Technology*, 49(5):483–492, 2007.

[120] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–355, London, UK, Jun. 2001.

[121] Kirk G. Fleming. We're Skewed - The Bias in Small Samples from Skewed Distributions. `http://www.casact.org/pubs/forum/07spforum/07Sp7.pdf`, 2007.

[122] B. Kitchenham, S. Pfleeger, and N. Fenton. Towards a Framework for Software Measurement Validation. *Software Engineering, IEEE Transactions on*, 21(12):929 – 944, Dec. 1995.

[123] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary Guidelines for Empirical Research in Software Engineering. *Software Engineering, IEEE Transactions on*, 28(8):721–734, Aug. 2002.

[124] B. A. Kitchenham. An Evaluation of Software Structure Metrics. In *Proceedings of the 12th International Computer Software and Application Conference (COMPSAC 1988)*, pages 369–376. IEEE Computer Society Press, 1988.

[125] D. E. Knuth. An Empirical Study of FORTRAN Programs. *Softw., Pract. Exper.*, 1(2):105–133, 1971.

[126] D. Koenig, A. Glover, P. King, G. Laforge, J. Skeet, and J. Gosling. *Groovy in Action.* manning publications Co., 2007.

[127] R. J. Koubek, G. Salvendy, H. E. Dunsmore, and W. K. LeBold. Cognitive issues in the process of software development: review and reappraisal. *International Journal of Man-Machine Studies*, 30(2):171–191, 1989.

[128] L. Lavazza and A. Agostini. Automated Measurement of UML Models: An Open Toolset Approach. *Journal of Object Technology*, 4(4):115–134, May 2005.

[129] M. Lehman. Laws of Software Evolution Revisited. In *Proceedings of the European Workshop on Software Process Technology*, pages 108–124, Berlin, 1996.

[130] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml System, Release 3.12. `http://caml.inria.fr/pub/docs/manual-ocaml`, 2011.

[131] W. Li and S. Henry. Object-Oriented Metrics That Predict Maintainability. *Journal of systems and software*, 23(2):111–122, 1993.

[132] K. Lieberherr and I. Holland. Assuring Good Style for Object-Oriented Programs. *Software, IEEE*, 6(5):38–48, Sept. 1989.

[133] E. Limpert, S. W. A., and M. Abbt. Log-normal Distributions across the Sciences: Keys and Clues. *BioScience*, 51(5):341–352, May 2001.

[134] R. Lincke, J. Lundberg, and W. Löwe. Comparing Software Metrics Tools. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 131–142, Seattle, WA, USA, 2008.

[135] M. O. Lorenz. Methods of Measuring the Concentration of Wealth. *Publications of the American Statistical Association*, 9(70):209–219, Jun. 1905.

[136] T. Love. An Experimental Investigation of the Effect of Program Structure on Program Understanding. *SIGPLAN Not.*, 12:105–113, Mar. 1977.

[137] M. Lumpe. Using Metadata Transformations to Integrate Class Extensions in an Existing Class Hierarchy. In N. Kobayashi, editor, *Proceedings of the Fourth ASIAN Symposium on Programming Languages and Systems (APLAS 2006)*, LNCS 4279, pages 290–306, Sydney, Australia, Nov. 2006.

[138] M. Lumpe. Growing a Language: The GLoo Perspective. In *Proceedings of the 7th international conference on Software composition*, SC'08, pages 1–19, Berlin, Heidelberg, 2008. Springer-Verlag.

[139] M. Lumpe, S. Mahmud, and O. Goloshchapova. jCT: A Java Code Tomograph. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 616–619, Lawrence, KS, USA, Nov. 2011.

[140] M. Lumpe, S. Mahmud, and R. Vasa. On the Use of Properties in Java Applications. In *Proceedings of the 21st Australian Software Engineering Conference*, pages 235–244, Auckland, New Zealand, Apr. 2010.

[141] D. Malayeri and J. Aldrich. Is Structural Subtyping Useful? An Empirical Study. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint*

European Conferences on Theory and Practice of Software, ETAPS 2009, pages 95–111, Berlin, Heidelberg, 2009.

[142] I. Maman and Y. Gil. Whiteoak: Introducing Structural Typing into Java. http://www.research.ibm.com/haifa/Workshops/ple2008/present\discretionary{-}{}{}/Whiteoak-ple08.pdf, Jun. 2008.

[143] D. Mancl and W. Havanas. A Study of the Impact of C++ on Software Maintenance. In *Conference on Software Maintenance*, pages 63 – 69, Nov. 1990.

[144] H. Melton and E. Tempero. An Empirical Study of Cycles Among Classes in Java. *Empirical Software Engineering*, 12:389–415, 2007.

[145] H. Melton and E. Tempero. Static Members and Cycles in Java Software. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 136 – 145, Sep. 2007.

[146] T. Mens and M. Lanza. A Graph-Based Metamodel for Object-Oriented Software Metrics. *Electronic Notes in Theoretical Computer Science*, 72(2):57–68, 2002.

[147] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-specific Languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.

[148] Metrics Plugin for Eclipse IDE. http://metrics.sourceforge.net/, 2010.

[149] M. Miles and A. Huberman. *Qualitative Data Analysis - An Expanded Sourcebook* . SAGE Publications, Inc, 1994.

[150] H. Mills. Software Engineering Education. *Proceedings of the IEEE*, 68(9):1158–1162, Sep. 1980.

[151] MISRA Ltd. MISRA-C: Guidelines for the Use of the C Language in Critical Systems, Oct. 2004.

[152] MISRA Ltd. MISRA-C++: Guidelines for the Use of the C++ Language in Critical Systems, 2008.

[153] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 2 edition, 2000.

[154] RSM Tool, M-Squared Technologies. http://msquaredtechnologies.com/, 2010.

[155] R. Muschevici, A. Potanin, E. Tempero, and J. Noble. Multiple Dispatch in Practice. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 563–582, 2008.

[156] E. Nasseri, S. Counsell, and M. Shepperd. Class Movement and Re-Location: an Empirical Study of Java Inheritance Evolution. *Journal of Systems and Software*, 2009.

[157] P. Naughton and H. Schildt. *Java 2: The Complete Reference*. McGraw-Hill, third edition, 1999.

[158] J. M. Neighbors. *Software Construction using Components*. PhD thesis, Department of Computer Science, University of California, Irvine, 1980.

[159] M. Newman. Power Laws, Pareto Distributions and Zipf's Law. *Contemporary physics*, 46(5):323–352, 2005.

[160] B. Nuseibeh and S. Easterbrook. Requirements Engineering: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, 2000.

[161] A. Offutt. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5–20, 1992.

[162] D. L. Parnas. Information distribution aspects of design methodology. In *IFIP Congress (1)*, pages 339–344, 1971.

[163] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[164] J. Parsons and Y. Wand. Using Objects for Systems Analysis. *Commun. ACM*, 40:104–110, Dec. 1997.

[165] J. Paulson, G. Succi, and A. Eberlein. An Empirical Study of Open-Source and Closed-Source Software Products. *IEEE Transactions on Software Engineering*, pages 246–256, 2004.

[166] D. E. Perry, A. A. Porter, and L. G. Votta. Empirical Studies of Software Engineering: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 345–355, Limerick, Ireland, 2000.

[167] L. Peterson and M. J. Peterson. Short-term Retention of Individual Verbal Items. *Journal of Experimental Psychology*, 58:193–198, 1959.

[168] S. Pfleeger, R. Jeffery, B. Curtis, and B. Kitchenham. Status Report on Software Measurement. *IEEE Software*, 14(2):33–43, Mar./Apr. 1997.

[169] S. L. Pfleeger. Experimental Design and Analysis in Software Engineering: Types of Experimental Design. *SIGSOFT Softw. Eng. Notes*, 20(2):14–16, Apr. 1995.

[170] G. Phipps. Comparing Observed Bug and Productivity Rates for Java and C++. *Softw. Pract. Exper.*, 29(4):345–358, Apr. 1999.

[171] C. Ponder and B. Bush. Polymorphism Considered Harmful. *SIGPLAN Not.*, 27(6):76–79, Jun. 1992.

[172] D. Posnett, V. Filkov, and P. T. Devanbu. Ecological Inference in Empirical Software Engineering. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 362–371, Lawrence, KS, USA, 2011.

[173] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-Free Geometry in OO Programs. *Communications of the ACM*, 48(5):99–103, May 2005.

[174] R. S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, Sixth edition, 2005.

[175] J. Project. JMetric - A Java Metric Analyzer. `http://jmetric.sourceforge.net/`, 2005.

[176] T.-S. Quah and M. M. T. Thwin. Application of neural networks for software quality prediction using object-oriented metrics. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 116–125, sept. 2003.

[177] Qualitas Research Group. Qualitas Corpus release-20101126. `http://qualitascorpus.com/docs/history/index.html`, Nov. 2010.

[178] D. R. Raymond. Reading Source Code. In *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '91, pages 3–16. IBM Press, 1991.

[179] E. Raymond. *The Cathedral & the Bazaar : Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly Media, Inc., 2008.

[180] P. Rechenberg. Programming Languages as Thought Models. *Structured Programming*, 11:105–115, 1990.

[181] W. Reed. The Pareto, Zipf and Other Power Laws. *Economics Letters*, 74(1):15–19, 2001.

[182] M. Reinhold. Project Lambda: Straw-Man Proposal. `http://cr.openjdk.java.net/~mr/lambda/straw-man`, 2009.

[183] A. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.

[184] J. R. Rose. Better Closures. `http://blogs.oracle.com/jrose/entry/better_closures`, 2006.

[185] N. Schneidewind. The State of Software Maintenance. *IEEE Transactions on Software Engineering*, SE-13(3):303–310, Mar. 1987.

[186] N. Schneidewind. Methodology for Validating Software Metrics. *IEEE Transactions on Software Engineering*, 18(5):410 – 422, May 1992.

[187] M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, Burlington, MA, 2009.

[188] C. Seaman. Qualitative Methods in Empirical Studies of Software Engineering. *Software Engineering, IEEE Transactions on*, 25(4):557 –572, Jul./Aug. 1999.

[189] S. S. Shapiro and M. B. Wilk. An Analysis of Variance Test for Normality (Complete Samples). *Biometrica*, 52(3/4):591–611, 1965.

[190] M. Shaw. Abstraction Techniques in Modern Programming Languages. *Software, IEEE*, 1(4):10 –26, Oct. 1984.

[191] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Apr. 1996.

[192] F. T. Sheldon, K. Jerath, and H. Chung. Metrics for maintainability of class inheritance hierarchies. *Journal of Software Maintenance*, 14:147–160, May 2002.

[193] B. Shneiderman. Exploratory Experiments in Programmer Behavior. *International Journal of Parallel Programming*, 5:123–143, 1976.

[194] F. Shull, J. Carver, S. Vegas, and N. Juristo. The Role of Replications in Empirical Software Engineering. *Empirical Software Engineering*, 13:211–218, 2008.

[195] M. Sikora. *Java Practical Guide for Programmers*. Burlington : Elsevier, 2002.

[196] M. E. Sime, T. R. G. Green, and D. J. Guest. Psychological Evaluation of Two Conditional Constructions Used in Computer Languages. *International Journal of Human-Computer Studies*, 51(2):125–133, 1999.

[197] D. Sjoberg, B. Anda, E. Arisholm, T. Dyba, M. Jorgensen, A. Karahasanovic, E. Koren, and M. Vokac. Conducting Realistic Experiments in Software Engineering. In *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n*, pages 17–26, 2002.

[198] I. Sommerville. *Software Engineering*. Addison Wesley, sixth edition, 2000.

[199] D. F. Specht. Probabilistic Neural Networks. *Neural Networks*, 3(1):109–118, 1990.

[200] M. Sridharan and A. S. Namin. Bayesian Methods for Data Analysis in Software Engineering. In *Proceedings of 32nd International Conference on Software Engineering, Volume 2*, pages 477–478, Cape Town, South Africa, May 2010.

[201] S. Stevens. On the Theory of Scales of Measurement. *Science*, 103(2684):677–680, 1946.

[202] R. Subramanyam and M. Krishnan. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.

[203] J. Sweller. Cognitive Load Theory, Learning Difficulty, and Instructional Design. *Learning and Instruction*, 4(4):295–312, 1994.

[204] T. Tamai and T. Nakatani. Analysis of software evolution processes using statistical distribution models. In *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE '02, pages 120–123, New York, NY, USA, 2002.

[205] M.-H. Tang, M.-H. Kao, and M.-H. Chen. An Empirical Study on Object-Oriented Metrics. In *Proceedings of the 6th International Symposium on Software Metrics*, pages 242–249, 1999.

[206] E. Tempero. An Empirical Study of Unused Design Decisions in Open Source Java Software. In *Proceedings of the 15th Asia Pacific Software Engineering Conference (APSEC)*, pages 33–40, Beijing, China, Dec. 2008.

[207] E. Tempero. How Fields are Used in Java: An Empirical Study. In *Proceedings of 20th Australian Software Engineering Conference (ASWEC'09)*, pages 91–100, Gold Coast, Queensland, 2009.

[208] E. Tempero, S. Counsell, and J. Noble. An Empirical Study of Overriding in Open Source Java. In *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science - Volume 102*, ACSC '10, pages 3–12, Darlinghurst, Australia, 2010.

[209] E. Tempero, J. Noble, and H. Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 667–691, Berlin, Heidelberg, 2008. Springer-Verlag.

[210] A. Tversky and D. Kahneman. Rational Choice and the Framing of Decisions. *Journal of business*, 59(S4):251, 1986.

[211] S. Valverde and R. V. Sole. Hierarchical Small-Worlds in Software Architecture. *ArXiv preprint cond-mat/0307278 - arxiv.org*, 2003.

[212] A. van Deursen, P. Klint, and J. Visser. Domain-specific Languages: An Annotated Bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[213] C. van Koten and A. Gray. An Application of Bayesian Network for Predicting Object-oriented Software Maintainability. *Information and Software Technology*, 48(1):59–67, 2006.

[214] G. van Rossum. Python Tutorial. `http://www.paradigma.com.br/Plone/tutorialPython.pdf`, 2003.

[215] P. Van-Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, 2004.

[216] R. Vasa. *Growth and Change Dynamics in Open Source Software Systems.* PhD thesis, Faculty of Information and Communication Technologies, Swinburne University of Technology, Oct. 2010.

[217] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative Analysis of Evolving Software Systems Using the Gini Coefficient. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'09).* IEEE Computer Society, 2009.

[218] R. Vasa, M. Lumpe, and A. Jones. Helix - Software Evolution Data Set. `http://http://www.ict.swin.edu.au/research/projects/helix`, 2010.

[219] R. Vasa, M. Lumpe, and J.-G. Schneider. Patterns of Component Evolution. In M. Lumpe and W. Vanderperren, editors, *Proceedings of the 6th International Symposium on Software Composition (SC)*, pages 244–260, Braga, Portugal, Mar. 2007.

[220] R. Vasa and J.-G. Schneider. Evolution of Cyclomatic Complexity in Object Oriented Software. In *Proceedings of the 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE '03)*, Darmstadt, Germany, Jul. 2003.

[221] B. Vasilescu, A. Serebrenik, and M. van den Brand. By No Means: A Study on Aggregating Software Metrics. *2nd International Workshop on Emerging Trends in Software Metrics*, 2011.

[222] B. Vasilescu, A. Serebrenik, and M. van den Brand. You Can't Control the Unfamiliar: A Study on the Relations Between Aggregation Techniques for Software Metrics. In *27th IEEE International Conference on Software Maintenance (ICSM)*, pages 313–322, Williamsburg, Virginia, USA, Sep. 2011.

[223] JHawk Java Metric Analyzer, Virtual Machinery. `http://www.virtualmachinery.com/jhawkprod.htm`, 2010.

[224] J. Voigt, W. Irwin, and N. Churcher. Class Encapsulation and Object Encapsulation - An Empirical Study. In *5th International Conference Evaluation of Novel Approaches to Software Engineering*, pages 171–178, Athens, Greece, Jul. 2010.

[225] C. Wang and D. Hou. An Empirical Study of Function Overloading in C++. In *8th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 47 – 56, Sep. 2008.

[226] L. Weissman. Psychological Complexity of Computer Programs: An Experimental Methodology. *SIGPLAN Not.*, 9(6):25–36, Jun. 1974.

[227] E. Weyuker. Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, Sep. 1988.

[228] R. Wheeldon and S. Counsell. Power Law Distributions in Class Relationships. In *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation*, pages 45–54, 2003.

[229] C. Wholin, P. Runeson, M. Höst, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction.* Kluwer Academic Publishers, 2000.

[230] L. E. Winslow. Programming Pedagogy - A Psychological Overview. *ACM SIGCSE Bulletin*, 28:17–22, Sep. 1996.

[231] C. Wohlin, M. Höst, and K. Henningsson. Empirical Research Methods in Software Engineering. In R. Conradi and A. Wang, editors, *Empirical Methods and Studies in Software Engineering*, volume 2765 of *Lecture Notes in Computer Science*, pages 7–23. Springer Berlin / Heidelberg, 2003.

[232] D. Wooff, M. Goldstein, and F. Coolen. Bayesian Graphical Models for Software Testing. *Software Engineering, IEEE Transactions on*, 28(5):510 –525, May 2002.

[233] H. K. Wright, M. Kim, and D. E. Perry. Validity Concerns in Software Engineering Research. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 411–414, New York, NY, USA, 2010.

[234] K. Xu. How Has the Literature on Gini's Index Evolved in the Past 80 Years? Department of Economics, Dalhouse University, Halifax, Nova Scotia, Dec. 2004.

[235] R. R. Young. Recommended Requirements Gathering Practices. *The Journal of Defense Software Engineering*, pages 9–12, Apr. 2002.

[236] E. A. Youngs. Human Errors in Programming. *International Journal of Man-Machine Studies*, 6(3):361–376, 1974.

Appendix A

Research Questions

We addressed various research questions in respective chapters. We present them in this section to gain an overview of all of them together. Research questions RQ1 to RQ5 are addressed in Chapter - 4 (Field Analysis), RQ6 to RQ9 in Chapter - 5 (Property Analysis), and RQ10 to RQ14 in Chapter - 7 (Inner Class Analysis).

- **RQ1**: What is the typical field distribution profile that developers usually practice? Do they follow available recommendations?
- **RQ2**: Does field distribution vary across different domains (e.g., middleware, database)? Do the results of RQ1 hold at domain level?
- **RQ3**: Do developers define fields in all classes in Java-based software systems? What is the typical distribution of field hosting classes and field inheriting classes?
- **RQ4**: Given a software system, are the volume and distribution of its fields correlated?
- **RQ5**: What is the typical profile of field exposure in Java-based software systems? Do developers confine exposed fields in a few classes or do they disperse them in almost every field hosting class?

- **RQ6**: What is the typical distribution profile of properties in Java-based software systems? Do developers adhere to the Java-specific coding conventions and associated guidelines while formulating solutions using getter and setter methods?

- **RQ7**: Is there any impact of the underlying problem domain on the use of getter and setter methods?

- **RQ8**: Do developers define getter and setter methods when they define private fields?

- **RQ9**: What is the distribution of ratio of getter and setter methods?

- **RQ10**: What is the typical distribution profile of inner classes in Java-based software systems? Does the use of inner classes vary across different domains (e.g., database, middleware)?

- **RQ11**: Do developers confine the use of inner classes to few classes only?

- **RQ12**: Do they use inheritance substantially to define inner classes?

- **RQ13**: Do they create highly nested abstractions using inner classes?

- **RQ14**: What is the typical distribution profile of methods in inner classes? Do SAM types occur with a high frequency?

Appendix B

jCT - Java Code Tomograph

B.1 Background

"Like physics, medicine, manufacturing, and many other disciplines, software engineering requires the same high level approach for evolving the knowledge of the discipline; the cycle of model building, experimentation, and learning" [31]. To facilitate experimentation, software engineering literature provides us with different guidelines [31,33,34,116,169,197, 229] that assist us to devise and conduct experiments to evolve the knowledge in field.

Measurement is the fundamental element here [40, 56, 77]. According to DeMarco [66] *"You cannot control what you cannot measure"*. To better understand the process being used to build a software system and the software system itself, we can identify a set of characterizing properties (or attributes), define associated software metrics, and perform an empirical analysis to interpret the data [40, 81]. These measurement activities can enable us to improve development processes and practices (e.g., [56, 140, 217]).

But conducting the desired measurement activity is a non-trivial task. It entails a number of elements that provide us with the fundamental ingredients for accomplishing a desired measurement task. These include the *program codes* that are intended to be measured, and matching *tool supports* to assist the measurement work. The former serves as the *basis* of meaningful information that assists us to bridge potential gap in knowledge, and the later provides us with the necessary *means* to accomplish the task mining information from the program codes. Though many different additional resources (e.g., data processing, aggregation, interpretation and reporting tools) can be involved in a measurement task, the experimental program code and data mining tools are the primary elements in studies entailing measurement tasks.

But the lack of availability and appropriateness of matching tool support often stand as barriers in conducting desired measurement and thus limits our capability. According to Tempero et al. [77] *"barriers to measuring code and understanding what the measurements mean include access to code to measure and the tools to do the measurement"*. Elimination of such barriers can increase our capability in conducting measurement effectively, and assist us to enrich our understanding on developers practices in software development and also the state of resulting software systems. But how can we eliminate such barriers?

Recently, researchers contributed different curated repositories of program code. For example, Tempero et al. [77] contributes the Qualitas Corpus and Vasa et al. [218] provides *Helix*. Both offer over 1200 releases of more than 100 open source Java-based software systems. But a curated collection of code is just one side of a coin. We need matching tool support for extracting desired metrics data from such repositories of code. Without appropriate tool support, it is difficult to capitalize the benefits arising from the available curated code repositories (e.g., Qualitas Corpus, Helix).

The set of functionality provided in a data mining tool is determined by the definition of software metrics that are embedded into the blueprint of a data mining tool. These entrenched metric definitions are used while processing experimental software systems to mine necessary metric data. The more metric definitions are embedded, the more metric can be extracted. But defining a finite set of metrics is a non trivial task as metric definitions are evolving over time.

Evolving Nature of Software Metrics

Software systems evolve over time [216]. As requirements are subject to change [96], software systems need to be updated to reflect them. According to Lehman's first law [129], a software system becomes less useful over time unless it is adapted continuously. As a result, they are required to pass through a process of continuous evolution to cope with growing demands on their functionality and usability.

The changing necessity of software systems causes not only the introduction of new programming language semantics over time, but also the corresponding design decisions of the associate developers to evolve. The new language semantics provides us with the desired means to capture evolving necessities, and the developers employ the new semantics to fabricate their design decisions. For example, the Java programming language has evolved over time and included many different new features. These include the concept of *inner class* (introduced in Java 1.1 [157]), *generics* and *enumerations* (introduced in Java 5.0).

Such evolving nature of programming language features, design decisions, and requirements of software systems are among the potential reasons that initiate new metrics definitions to emerge over time. During the last few decades, a variety of software metrics definition sets have been proposed [28, 35, 55, 56, 99] in order to measure different aspects of software systems, and also the underlying design decisions [77, 77, 140, 209]. Though some of them are widely used (e.g., [56]), no set of metrics definition has yet been identified as *standard*, causing software metrics to emerge over time.

Metrics Requirement in Empirical Study

An empirical study can rely on different and disjoint set of software metrics depending on the associated objectives. We classify them into following three categories:

1. **Existing software metrics**
   A study can employ only the available software metrics without introducing any new set of metrics of its own. For example, Subramanyam and Krishnan [202] investigated the CK metrics suite [56]- an available set of metrics, for determining software defects.

2. **Combination of both existing and new metrics**
   A study can use both the existing metrics and define new set of metrics based on the associated requirements. For example, Tang et al. [205] used a combination of available metrics (i.e., CK metrics suite [56]) and new set of software metrics (e.g., coupling between methods, average method complexity, inheritance coupling, and number of object allocation) to investigate the correlation between object-oriented design metrics and the likelihood faults.

3. **New set of metrics**
   A study can completely introduce a new set of metrics. For example, this work requires new sets of software metrics (e.g., properties, inner classes) to investigate developers behaviors and decisions in Java-based software systems.

Either way, matching tool supports are required to extract the desired software metrics data. While the availability of fitting tools provides us with the means to extract desired software metrics data for goal attainment, the lack of appropriate tool supports can substantially limit the flexibility of necessary software metrics data mining task, and can often prevent us to proceed (as goal attainment in an empirical study significantly depends on available metrics data).

B.2 Available Approaches for Metrics Data - Mining

In software engineering literature, there are different approaches available for supporting software metrics data mining tasks. We classify the approaches into: (i) Metrics-driven approach , and (ii) Meta model-based approach.

1. Metrics-driven Approach

In the metrics driven approach, certain predefined metrics definitions are embedded in tools. This approach is found to be adopted in both open source (e.g., [2,3,8,9,23,58,115,148]) and commercial tools (e.g., [14,154,223]). The formers are free to use and modify as necessary and the later are restricted by certain copyrights.

As tools built on the metrics driven approach can extract only the metrics they are designed for, their applicability and usability in projects requiring slightly different set of metrics is fairly limited. Therefore, often they may not satisfy user expectation completely.

Table B.1: Tools that rely on Metric-driven Approach

Tool	Description
JMT [2]	Supports two specific modes: single and multiple files processing limited to 19 predefined metrics definitions. Does not provide any API for incorporating new software metrics.
JHawk [223]	Jhawk comes in three flavors : stand alone application (both GUI and Command Line) and Eclipse plugin. Provides metrics in system, package, class and method level. Does not provide any API for incorporating new software metrics.
Metrics [148]	It is in fact a plugin for the Eclipse platform. It can compute metrics and detect cycles in package and type dependency graph. No support for incorporating new software metrics.
Dependency Finder [6]	Primarily focused on dependency analysis but also include metrics extraction functionality. Only certain object oriented metrics can be extracted. No support for incorporating new software metrics.
JDepend [8]	Computes 9 pre-defined metrics at package level only. Does not provide any API for incorporating new software metrics.

Consider, for example, a snapshot of the use of property mechanism available in Java programming language. This study requires a set of new metrics that capture different aspects of the property mechanism (e.g., definitions of getter and setter methods and their variants). As available tool support does not offer the desired metrics, we can have four potential options described in Figure B.1.

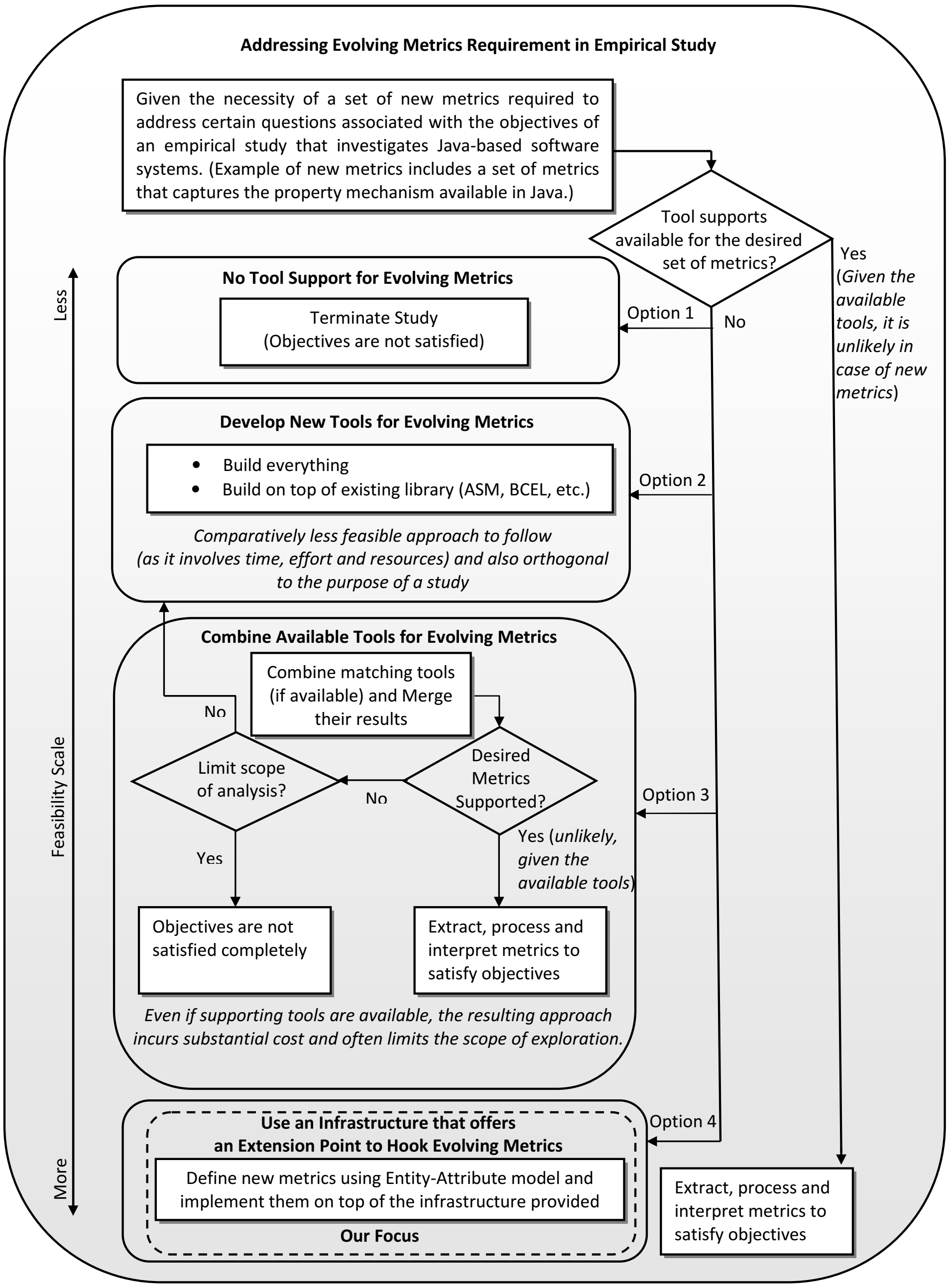

Addressing Evolving Metrics Requirement in Empirical Study
Given the necessity of a set of new metrics required to address certain questions associated with the objectives of an empirical study that investigates Java-based software systems. (Example of new metrics includes a set of metrics that captures the property mechanism available in Java.)
Tool supports available for the desired set of metrics?
Yes (Given the available tools, it is unlikely in case of new metrics)
No
Option 1
No Tool Support for Evolving Metrics
Terminate Study (Objectives are not satisfied)
Option 2
Develop New Tools for Evolving Metrics
• Build everything
• Build on top of existing library (ASM, BCEL, etc.)
Comparatively less feasible approach to follow (as it involves time, effort and resources) and also orthogonal to the purpose of a study
Option 3
Combine Available Tools for Evolving Metrics
Combine matching tools (if available) and Merge their results
Desired Metrics Supported?
No
Limit scope of analysis?
No
Yes
Yes (unlikely, given the available tools)
Objectives are not satisfied completely
Extract, process and interpret metrics to satisfy objectives
Even if supporting tools are available, the resulting approach incurs substantial cost and often limits the scope of exploration.
Option 4
Use an Infrastructure that offers an Extension Point to Hook Evolving Metrics
Define new metrics using Entity-Attribute model and implement them on top of the infrastructure provided
Our Focus
Extract, process and interpret metrics to satisfy objectives
Less
Feasibility Scale
More

Figure B.1: Addressing Evolving Metrics requirement in an Empirical Study

- Terminate the study due to the lack of available tool supports, resulting in the objectives of the study to remain unattained.
- Develop new tools either from scratch or build the necessary component on top of the available libraries (e.g., ASM, BCEL). The decision to build tools to address research questions often becomes infeasible as it involves a time, effort and resources. More importantly, building tools often becomes orthogonal to the original purpose of the study.
- A third potential option can be combining matching tool sets to accomplish the desired task by merging their results together to address the necessities. But such approach, if supported tools are available, often limit the scope of analysis and also may not lead to fulfilling the exact requirement of the study.
- A fourth potential option can be to use an infrastructure that provides necessary supports for incorporating new metrics definition on top of the existing facilities provided.

Therefore, it becomes a non-trivial task to work with existing tool support when a particular study demands new set of software metrics. Moreover, the available tools often produce differing outcomes for same input. A recent study [134] concludes that *"there are differences between the metrics measured by different tools given the same input."* and *"it (the difference) does matter and might lead to different conclusions"*.

2. Meta Model-based Approach

A meta model-based approach, on the other hand, is concerned with formulating generic (often language independent) model for metrics definitions. There are many different studies (cf. Table B.2) that considered the development of a common meta model using various techniques (e.g., construction of relational database schema for storing metadata). A common theme of such studies is once the model is constructed, different query languages like Object Constraint Language (OCL), XML Query Language (XQuery), Structured Query Language (SQL) are being used for computing desired software metrics.

Table B.2: Meta Model-based Approaches

Example of Different Approaches		
Study	Description	Note/Limitation
Harmer and Wilkie [97]	Meta model is defined in the form of relation database schema and metrics are computed using SQL.	Defining metrics is a non-trivial task. Complex metric (e.g., CK's CBO metric) requires embedded SQL in program code written in C.
Baroni and Abreu [29]	OCL is used to define metrics.	Though design related metrics can be expressed using OCL (on UML class diagram), metrics that demand implementation-specific data is difficult to define.
Wakil et al. [70]	Describes an approach to express metrics in terms of XQuery expressions on UML Model.	Metrics implementation requires complex XQuery code.
Lavazza et al. [128]	Defines relational database tailored for storing UML model data	Supports design level (i.e., UML-based) metrics (*e.g.*, CK metrics).

There is another family of tools [9, 115] that are constructed based on a number of light-weight meta model based libraries (ASM [46], BCEL [5], and Javassist [53]). Though these libraries have not been designed with metrics extraction in mind, they can be used for this purposes. For example, while ASM is used as class file parser in JSeat and Mutations, BCEL and ASM provide the foundations for the static Java code analyzer FindBugs^{TM} [7]. The actual domain of these libraries is code instrumentation and the support for aspect-oriented programming [120]. The corresponding meta models permit them to work with real-world systems. However, they omit some details about the underlying language semantics that makes them less suitable for high-precision data mining.

Moreover, the construction of a meta-model often imposes constraints on the capability of certain tools. For example, both of JMetric [175] and JSeat [115] employ a meta model to map to the object-oriented language semantics of Java. The construction of the necessary model data exceeds the available runtime memory when they attempt to parse large software systems like Eclipse or Netbeans. In fact, an early version of JMetric could only cope with systems containing 5,000 classes or less. To reduce the actual memory footprint we can use some form of caching strategy to off-load data to a persistent storage (used in the latest version of JSeat). But this creates engineering challenges and maintenance overheads, usually orthogonal to the purpose of the tool itself.

Even though the use of meta models can limit the scope of exploration, we find them in many metrics-based software engineering approaches. For example, Mens and Lanza [146] pointed out that *language interference* can result in conflicting definitions for one and the same measure. To steer clear of this problem they developed a graph-based language-independent meta model for metrics definition. In this approach, nodes represent the language entities (i.e., classes, methods, statements, method invocations, etc.), whereas edges capture the relationships between nodes (*e.g.*, contains, overrides, accesses, etc.). The graph-based meta model provides three built-in *generic* measures (i.e., *Node Count*, *Edge Count*, and *Path Length*) that serve as the main building blocks to define, for example, object-oriented metrics. This approach allows for reusing existing metrics extraction infrastructures even in the face of newly emerging measures. However, this comes at a price. Many complex measures (*e.g.*, CFO or RFC [56]) cannot be expressed [146].

B.3 Our Approach

Our approach considers language features as the key to metrics extraction and processing. In this approach, language features come first, then the necessary metrics. The model of our approach (cf. Figure B.2) comprises three key components: (i) Task manager , (ii) Metric extractor, and (iii) Language-specific parser.

Task Manager

An empirical study involving analysis of contemporary software systems can often require system specific information. To reveal such information, one must examine the components that constitute the core functionality of a software system. But contemporary software systems, in general, make extensive use of third party libraries. Therefore, it is necessary to identify the desired set of artifacts. According to Tempero et al. [77], *"decisions need to be made regarding exactly what is going to be analyzed"*.

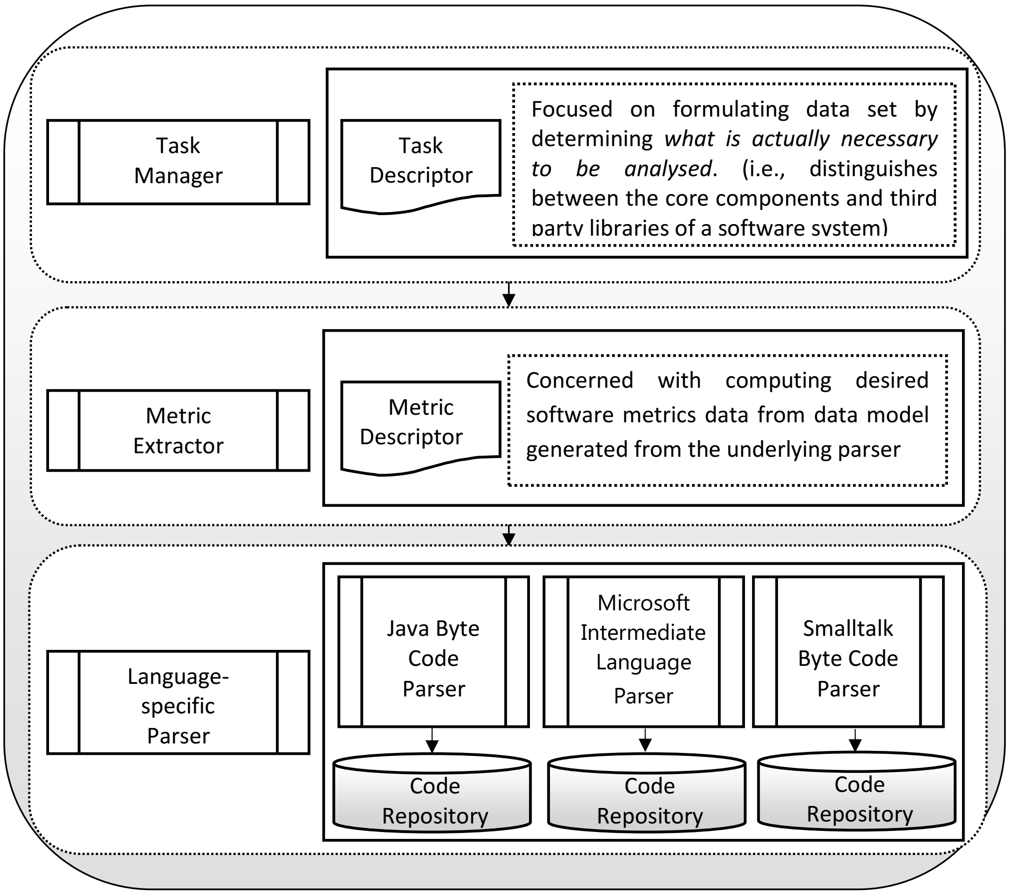

Task Manager
Task Descriptor
Focused on formulating data set by determining *what is actually necessary to be analysed*. (i.e., distinguishes between the core components and third party libraries of a software system)
Metric Extractor
Metric Descriptor
Concerned with computing desired software metrics data from data model generated from the underlying parser
Language-specific Parser
Java Byte Code Parser
Microsoft Intermediate Language Parser
Smalltalk Byte Code Parser
Code Repository
Code Repository
Code Repository

Figure B.2: Multi-layered Model for Software Metrics Extraction

Deciding on what need to be analyzed is just one side of a coin. The decision has to be reflected in the analysis process to formulate a fine-grained experimental software artifacts. Often it becomes substantially cumbersome to conduct large scale empirical study due to the lack of proper means to accomplish such task (i.e., *what should be included in analysis* and *what should not*). It is, therefore, beneficial to have a mechanism that can provide us with suitable set of artifacts based on that decision.

The task manager is equipped with such mechanism. To manage different task-specific processes, the task manager provides the necessary *meta data* for constructing *task-specific* data set based on task descriptor (e.g., configuration file). The task descriptor cooperates with the task manager to identify the *core* components of a software systems.

Metrics Extractor

The *Metric Extractor* adopts a *class-by-class* metric extraction approach. It accomplish its task in two steps: Firstly, it scans each *class* available in the data set, and then it applies the desired measures on each of them. In order to perform the first step, the *Metric Extractor* uses a *Class File Parser* (cf. Figure B.3).

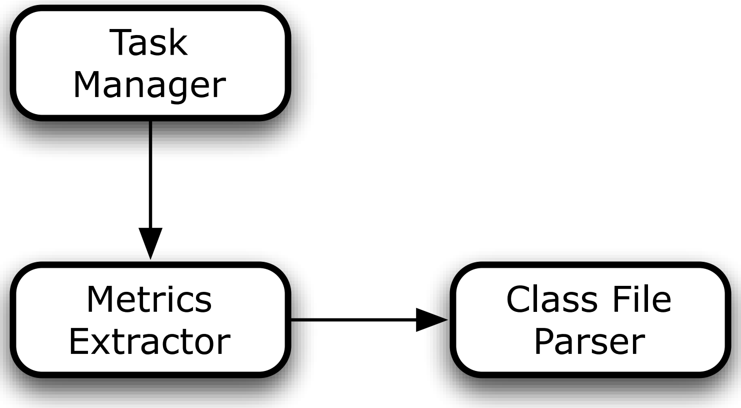


Figure B.3: jCT's main components and their interaction.

Language-specific Parser

The semantics of a language offers the most precise representation [137]. Therefore, we decide to use the semantics of a language (see Figure B.3) as the source of metric data mining. The semantics can be exploited from either the *source code* or *compiled code* of the software systems. In the first case, the source must be available to the user involved in data mining. This approach often becomes difficult unless the software artifacts are non-commercial, open source products. Thus, it restricts the selection of data set for analysis. An alternative way is to use the compiled code (which should be available regardless commercial interest of the artifacts) of the desired software artifacts. To parse compiled code of given language, we employ a language specific parser.

General	Fully-qualified class name Super class name Interfaces implemented Modifiers Constant Pool: numeric, string, and type constants Annotation* Attribute*
Field*	Modifiers, name, and type Annotation* Attribute*
Method*	Modifiers, name, return type, and parameter types Annotation* Attribute* (including Java IL-bytecode)

Table B.3: Structure of a compiled Java Class [47] (* indicates cardinality ≥ 0).

The *Class File Parser* is responsible for parsing a given *class file*, and providing a *JavaClassFile* instance for each of them. The *JavaClassFile* is fabricated to capture all the information available (see Table - B.3) in a Java class file.

The next step is concerned with distilling the desired measures from each of the class files. For this purpose, an on demand access to the underlying byte code instructions of a given class file is necessary. Byte codes are stored as a sequence of binary numbers in a class file [47]. In order to interpret them, these numbers have to be converted into the instances of jCT-byte code instructions - the internal jCT representation of Java byte code instruction. This interpretation process is very expensive in terms of memory consumption. To optimize the memory usage profile, jCT adopts a mechanism practiced usually in graphical user interface programming or operating systems. According to this mechanism, the bytecodes of particular class are treated as *resource* which is loaded and locked only on demand. After the required measures are distilled from a class, the byte codes are released to reduce jCT's runtime memory footprint.

B.4 jCT

jCT is a stand-alone Java application. In this section, we discuss jCT's main features, rationale for the associated design choice, flexibility of incorporating evolving measures, on board mechanism for interaction with the curated repositories, and runtime statistics to demonstrate its time and memory consumption profile.

B.4.1 Workflow

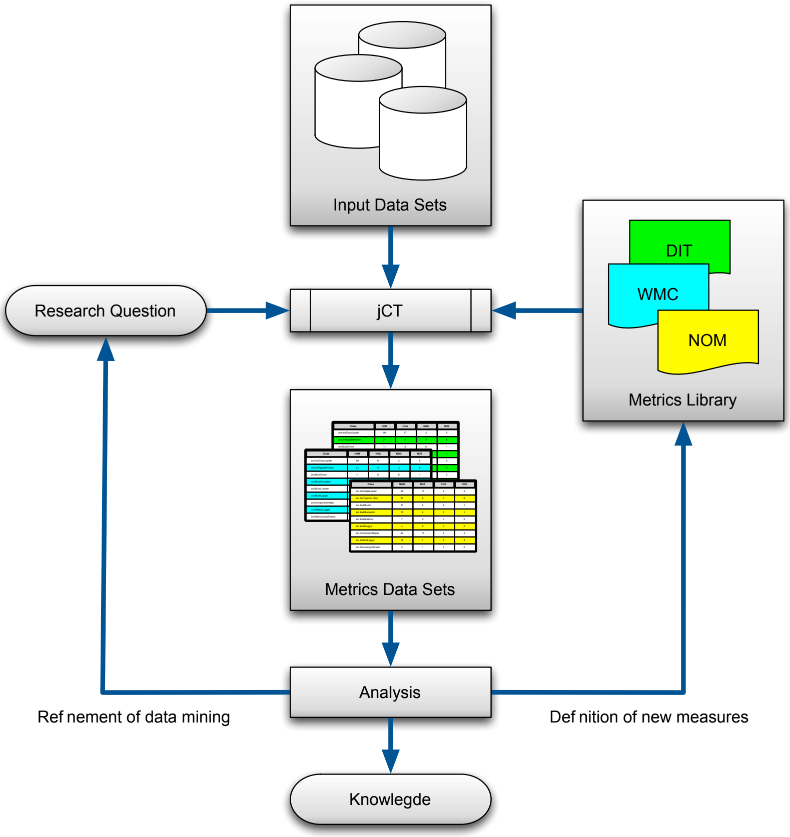


Figure B.4: jCT aided approach for software analysis

An empirical research project, in general, starts with a goal upfront. This goal is decomposed into matching set of research questions. Answering these questions demands appropriate set of metrics that are required to be mined from the associated software artifacts under investigation. But as empirical studies are exploratory in nature, often the necessity to investigate more promising aspects of software systems arises. To cope with such evolving necessities, it is required to refine research questions and corresponding set of metrics definitions.

jCT allow us not only to reflect such refinements, but also to incorporate new metrics definitions through offering extensible metrics data mining infrastructure. Thus jCT supports empirical research to enrich the current state of knowledge. The workflow diagram depicted in Figure B.4.

B.4.2 Task-based Metrics Data Mining

Empirical research work entails diverse set of dimensions, depending on the project-specific objectives. For example, a research project can seek to answer the *evolutionary* trends of a set features in a particular software system, and on the other hand, a different project can be interested in *comparative* analysis of a particular feature across different software systems. Such wide variety of activities involve not only different tasks, but also demand the use of completely different data sets.

Such diverse nature of work not only demands a variety of metrics but also involves various data sets. For example, the data set involve in analyzing single software system as a case study is completely different from the data set involved in the comparative study of hundreds of software systems.

However, processing such diverse data sets efficiently is a challenging task. One of the tactics to cope with such challenge is dividing the data mining activity into several *task-specific* approaches. More precisely, the divide and conquer strategy is applied on the task management activity. The resulting benefits include the underlying process of mining necessary metrics data is governed by only the associated task, and thus offer better efficiency in data processing with increased time and memory space utilization trade-offs.

To facilitate metrics data mining, jCT considers the following key tasks:

- **Single Task** - processing separate **.class* and **.jar* files;

- **Source-Item Task** - processing separate **.class* and **.jar* files from the specified source directory;

- **Bulk Task** - processing systems as a bulk from the root directory;

- **Qualitas Corpus Task** - processing systems from Qualitas Corpus data set as a bulk from the root directory using *.properties* file.

- **Helix Task** - processing systems from Helix data set for studying software evolution.

B.4.3 Support for New Measures

jCT offers the facility to weave new measures by employing an extraction engine that provides necessary supports for incorporating the desired metric definition. jCT adopts a light-weight extension mechanism for capturing new metric definition. This mechanism is *viewpoint-agnostic*, that is, the definition of new measures are not required to adhere to any predefined analysis model. This yields the facility to tailor every measure to the specific needs of the intended analysis. For example, jCT can be used not only to capture a wide variety of (both existing and new) object-oriented metrics, but also to emulate *javap* to build an independent Java class file disassembler. In addition, jCT's on board mechanism for emitting separate class and method graphs offers the flexibility of conducting independent graph-based analyses.

The jCT metrics extraction engine uses a 2-pass tactic in order to allow for the resolution of mutual dependencies between classes. By default, we always have to implement the first pass extraction method. The second pass provides an empty default behavior and only needs to be overridden when a measure requires it. For example, computing the CBO metric [56] entails the first pass to construct a the *type dependency graph* of the software system being analyzed, and the second pass can compute the CBO metric for each class based on the graph constructed during first pass.

jCT's extraction engine supports the *separation-of-concern* for metric definitions - an approach that enables to manipulate only the relevant activities together by separating them based on their purpose. Each newly defined measures is unique, and therefore, should be weaved into jCT independently. For this purpose, jCT's extraction engine is equipped with a hook class `MetricCollector` - an extension point for the definition of new measures. The key steps to incorporate a new metric definition to jCT is depicted in Figure B.5.

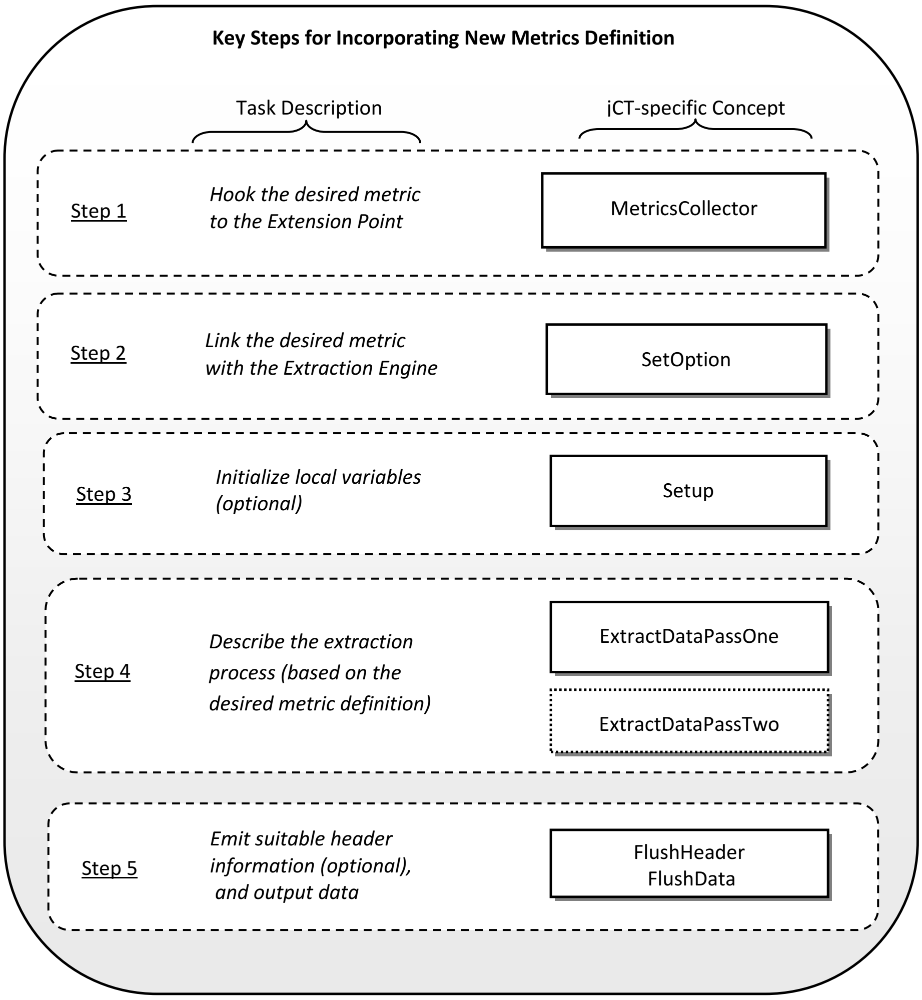

Key Steps for Incorporating New Metrics Definition
Task Description
jCT-specific Concept
Step 1
Hook the desired metric to the Extension Point
MetricsCollector
Step 2
Link the desired metric with the Extraction Engine
SetOption
Step 3
Initialize local variables (optional)
Setup
Step 4
Describe the extraction process (based on the desired metric definition)
ExtractDataPassOne
ExtractDataPassTwo
Step 5
Emit suitable header information (optional), and output data
FlushHeader
FlushData

Figure B.5: Key Steps to Incorporate New Metrics Definition in JCT

The definition of new measures has to describe the recording process of the attributes (required for computing the measures) per class. To facilitate this process, the `MetricCollector` offers the required infrastructure by providing it with six pre-defined methods - that can be customized to accomplish the desired task. The recording process might require the facility to initialize any local variables, if necessary. This is provided by the method `setup`. The `flushHeader` and `flushData` emit any header information and the computed metric output dataset, respectively.

The method `setOption` is responsible for hooking the defined measure with the extraction engine, and providing appropriate command line parameter for recognizing the new measure. jCT employs a configuration manager for registering all the available metric definitions with its extraction engine. These metrics are instantiated while the extraction engine boots up. Once the desired measure is plugged-in, it can be accessed by the provided parameter to commence the recording process.

B.4.4 Interaction With Curated Repositories

Software measures have to be properly *validated*. Basili et al. [32] demand that software metrics need to be *validated* in order the demonstrate their usefulness. The process of metrics validation has been addressed in several studies [122, 186, 227]. Schneidewind [186] characterized six validity criteria: association, consistency, discriminative power, tracking, predictability, and repeatability. The last criterion, metric validation by repeatability, demands a sufficient number of repeated studies in order to achieve enough confidence on a particular measure to demonstrate its effectiveness in signaling the level of quality of software artifacts. But conducting repeated empirical studies for such validation task is a non-trivial task. It demands a suitable collection of software artifacts, and a compatible data mining framework as pre-requisite.

The availability of a large set of open-source software has made this task easier. But collecting an appropriate set of software systems that covers a wide variety of *domain*s is a non-trivial task. A biased input data set might lead to obtaining misleading outcome. Special care, therefore, must be taken in order to select a right set of *representative* software systems for experimentation. The Qualitas Corpus [77] and Helix [218], two recently emerged *curated* open-source software repositories, aim exactly at this problem. Both offer combindly over 1200 releases of more than 100 open source Java based software systems. A wide variety of studies [140, 216] have already adopted them. Both come with the promise to enable equally repeatable large-scale empirical studies of code and the validation of software metrics in general.

The curated repositories are just one side of a coin. The lack of a suitable data mining framework that provides built-in supports for seamless extraction of desired metric data from those repositories prevents us from achieving the intended goal. jCT is equipped with *on-board mechanism* for coping up with such large curated repositories. It can distill desired metric data either from an entire repository at once or offers the facility to support system by system metric extraction (including large data set like *netbeans-6.9.1* comprising 34,368 classes). These functionalities yield the versatility necessary for both comparative and evolutionary studies.

B.4.5 Runtime Profile

The design choice of jCT is governed by a set of well defined criteria that aim at optimizing its runtime performance. This drives to adopt a *non meta model* based tactic. This is significantly influenced by the incompetence of earlier light weight meta model based tools (e.g., JSeat [115], JMetric [175]) that cannot cope up with larger systems. These become fridged while processing large data set like Netbeans, Eclipse or JBoss comprising many thousands of classes. JSeat employs the ASM bytecode engineering library [46], and uses a multi-pass parsing approach to parse Java class files. It has to rely on secondary persistent storage system for dumping partially processed data. The final report generation process seeks to load back them again. The entire process demands the consumption of extensive runtime memory. An early version of *JMetric* cannot handle software systems comprising more than $5,000$ classes. Therefore, optimizing runtime resource consumption to facilitate on the fly processing of larger systems is at the center of the design focus.

What is the runtime performance profile of jCT? How does it behave while processing large systems? To uncover the answers, we aim at measuring its time and memory consumption statistics during runtime. A commonly adopted method of performance profiling is to drive a laboratory experiment with a standard and well accepted set of population (i.e., software systems). The contributing entities of this set should represent a wide variety of domains (e.g., Database, SDK, Middleware, Games, Tools, etc.). This empowers us to unfold the inherent consequence of diverse population on jCT. Thus, the likelihood of any bias induced by a dominating domain is reduced. In addition, the data set should be composed of applications that are scattered well in their size dimension. This assists in establishing a revealing relation to get an insight into the memory consumptions by population of varying size. Besides, the experiment has to be equipped with a well defined suite of metrics. These must cover almost every aspects of the constructs associated with the Java programming language. This seeks to unveil the behavior of jCT under varying nature of mining tasks associated with each participating measure. Thus, the appropriate rationale forms the required ground for both the population and metric suite, and offers us with a means to characterize the runtime profile of jCT.

For this purpose, we use Qualitas Corpus version 20101126r [77]. We extract 140[1] different object-oriented class, method, and field measures relating to both size and complexity (*e.g.*, *Number of Methods* [217], *Number of Getters* [140], or *Response for a Class* [56]).

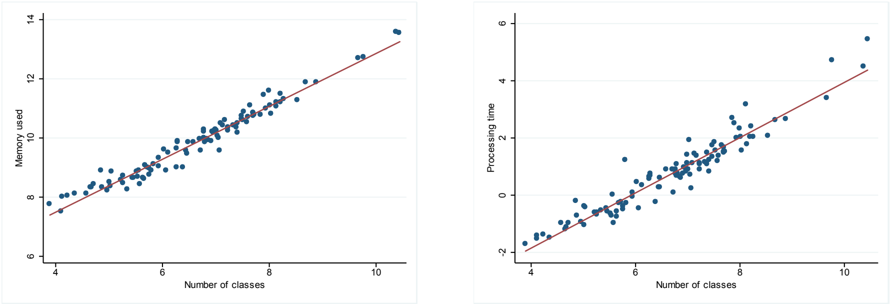
To conduct this laboratory test, we set up an experimental environment with a Mac Pro empowered by one 2.66 GHz Quad-Core processor, 8GB 1066 MHz DDR3 memory, and running Mac OS X 10.6.6. We employ jCT to run in this context assigning the task to mine the predefined metric suite from the provided population. The runtime scenario reveals that only 14 built-in Java and jCT types consume 80% of the runtime memory - strings (i.e., `char[]`) occupying more than 30% alone. String naturally is a memory intensive data type. We cannot afford to implement a design without employing minimal strings, and therefore, the design has to seek for reducing the cost incurred. jCT employs a dedicated string heap to adhere to one of the design criteria - optimize memory consumption. This string heap grows in memory as metric extraction task proceeds for a particular software system. Once a job is completed, it is cleared to remove unused strings from memory before another system from the pipeline enters into the execution engine.

[1] We record separate counts for storage and visibility modifiers. For example, the *Number of Methods* of a class yields 24 sub-measures: 6 for **static** methods, 6 for instance methods, 6 for **abstract static** methods, and 6 for **abstract** instance methods. It is this detailed breakdown that allows for a fine-tuned analysis of developer behavior [140, 217].

Runtime Object Allocation						
	Active Instances			Total Instances		
Rank	**Type**	**Count**	**Size in %**	**Type**	**Count**	**Size in %**
1	`char[]`	56	34.59	`char[]`	56	31.51
2	`ConstantUtf8`	56	8.53	`int[]`	56	24.34
	`RedBlackTree.RedBlackNode`	3	8.07			
3	`byte[]`	52	6.68	`ILInstruction`	56	12.67
4	`LineNumberInfo`	37	5.94	`ILArgument`	56	5.58
	`LocalVariableInfo`	33	5.17			
5	`ContantNameAndType`	13	4.66	`ILShortIndexToConstantPoolArgument`	45	4.22
	`ConstantPoolEntry[]`	29	4.48	`EntryIterator`	8	3.20

Figure B.6: Memory profiling statistics.

The memory profiling statistics of metrics extraction with jCT is depicted in Figure B.6. The presented data reveals two important aspects, *Active instance*, and *Total instance* of object allocation profile. The most memory-intensive active instances are comprised of class file (e.g., `ConstantUtf8`) and dataset (e.g., `RedBlackTree.RedBlackNode`) related entities. That is, only the data needed to represent Java classes in jCT is kept in memory at all times, On the other hand, the most memory-intensive total instances are associated the light-weight meta model for the analysis of IL-bytecode (i.e., the internal jCT representation of the instruction stream). However, these instances are released as soon as the metrics extraction for the host class has been completed, hence easing the cumulative demand on memory for metrics extraction in jCT.



(a) Memory consumption per system size(log$_e$ scale)

(b) Processing time per system size (log$_e$ scale)

Figure B.7: Runtime Statistics.

jCT's design aims at maintaining a linear time and space complexity while extracting metric data. The runtime profile depicted in Figure B.7 demonstrates that it yields $O(n)$. Both memory consumption and running time are proportional to the size of the system being analyzed. We observe a strong positive correlation between system size, in terms of number of classes, and memory consumption (i.e., $\rho = 0.98$) and running time (i.e., $\rho = 0.96$). The linear fits for memory consumption and running time are shown in Figure B.7(a) and Figure B.7(b), respectively.

B.5 Summary

We introduced a data mining framework, jCT - that facilitates extraction of not only existing software metrics, but also the emerging ones by providing the necessary means to incorporate new metrics definitions into the infrastructure through suitable extension point. jCT improves the flexibility of conducting empirical studies as it does not confined itself to any predefined set of measures, and eliminates the engineering challenges associated with necessary software metric data mining tasks.

Moreover, we discussed the underlying rationale of different aspects of this infrastructure (e.g., underlying motivating factors for design decisions, architecture, workflow). We showed jCT's capability to cope easily with large scale data set while maintaining linear time and space complexity in the extraction process.

jCT can support us by providing simple yet effective means for conducting large scale empirical studies, and allow us to advance the current understanding of programming language feature usage patterns in particular, and software engineering in general.

Appendix C

Analyzing Qualitas Corpus using jCT - An Example

jCT has built-in support for mining necessary software metrics from recently emerged curated repositories such as Qualitas Corpus and Helix. To demonstrate how jCT interacts with them, we present an example to extract the metrics: *Getters/Setters Variations* [140], *Number of Methods* [217], and the CK metrics suite [56] from the Qualitas Corpus.

```
java —Xms512m —Xmx1024m jct.Main —qc ../../QCInput
 —verbose —log log  —output ../../Output
 —gssummary gssummary —nom nom —ck ck
```

Figure C.1 shows the structure of input and output directories for Qualitas Corpus analysis.

Post Processing Raw Data

jCT produces raw data. Once all data has been mined, it can be post-processed to retrieve the necessary information, to perform statistical analysis. Figure C.2 shows an overview of possible post-processing activities.

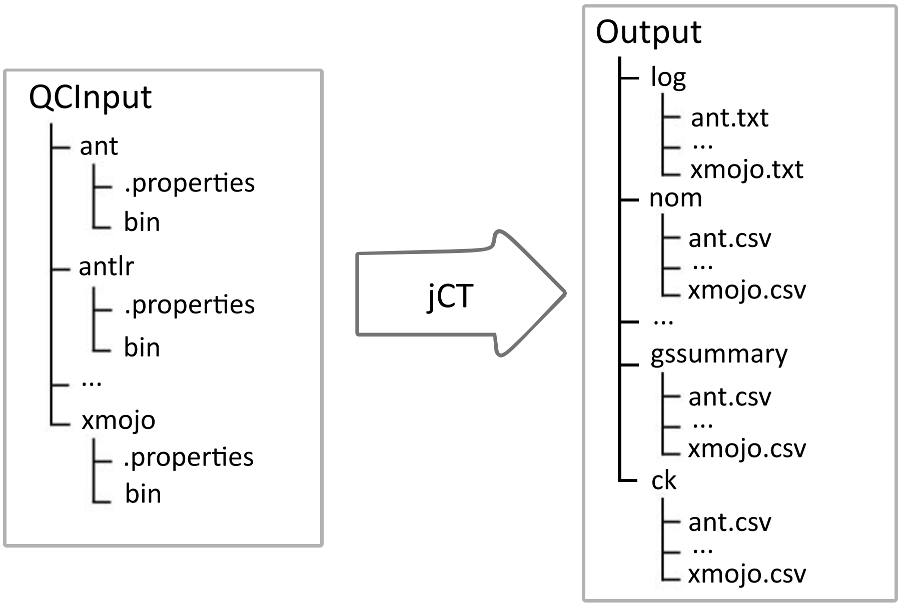
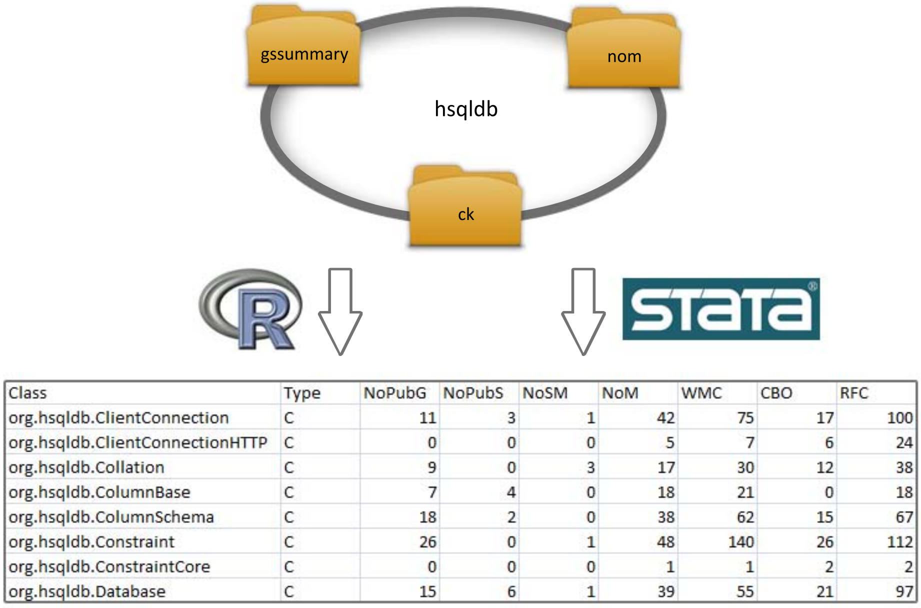


Figure C.1: Processing Qualitas Corpus.



Class	Type	NoPubG	NoPubS	NoSM	NoM	WMC	CBO	RFC
org.hsqldb.ClientConnection	C	11	3	1	42	75	17	100
org.hsqldb.ClientConnectionHTTP	C	0	0	0	5	7	6	24
org.hsqldb.Collation	C	9	0	3	17	30	12	38
org.hsqldb.ColumnBase	C	7	4	0	18	21	0	18
org.hsqldb.ColumnSchema	C	18	2	0	38	62	15	67
org.hsqldb.Constraint	C	26	0	1	48	140	26	112
org.hsqldb.ConstraintCore	C	0	0	0	1	1	2	2
org.hsqldb.Database	C	15	6	1	39	55	21	97

Figure C.2: HSQLDB Post Processing.

Appendix D

Raw Metric Data

We provide the raw metrics data used in this study (in the attached DVD). The data set is divided into three categories: field analysis (located in `MetricData/FieldAnalysis`), property analysis (located in `MetricData/PropertyAnalysis`), and inner class analysis (located in `MetricData/InnerClassAnalysis`). The metrics data for each software system is in a file with the format *SystemName-Version.csv*.

We provide illustrative samples of fields, properties and inner classes data of the system *apache ant-1.8.1* in listing D.1, D.2, and D.3, respectively. The first line is the header, and each of the following lines is a class along with its attributes as indicated by the associated header.

```
Class,Type,NoGenSA,NoSA,NoPubSA,NoProSA,NoPriSA,NoDefSA,NoGenA,NoA,NoPubA,NoProA,NoPriA,NoDefA
org.apache.tools.ant.AntClassLoader,C,4,6,0,0,6,0,0,10,0,0,10,0
org.apache.tools.ant.AntClassLoader$ResourceEnumeration,C,0,0,0,0,0,0,1,3,0,0,3,0
(more data...)
```

Listing D.1: Field Data - ant-1.8.1 (sample only)

```
Class,DataHolderCategory,NoPubG,NoProG,NoPriG,NoDefG,NoPubRG,NoProRG,NoPriRG,NoDefRG,NoPubVG,NoProVG,NoPriVG,NoDefVG,NoPubPG,NoProPG,NoPriPG,NoDefPG,NoPubS,NoProS,NoPriS,NoDefS,NoPubRS,NoProRS,NoPriRS,NoDefRS,NoPubVS,NoProVS,NoPriVS,NoDefVS,NoPubPS,NoProPS,NoPriPS,NoDefPS,NoPubBP,NoProBP,NoPriBP,NoDefBP,NoPubNBP,NoProNBP,NoPriNBP,NoDefNBP,NoPubGLM,NoProGLM,NoPriGLM,NoDefGLM,NoPubSLM,NoProSLM,NoPriSLM,NoDefSLM
org.apache.tools.ant.AntClassLoader,0,2,0,1,0,0,0,0,0,1,0,1,0,1,0,0,0,5,0,0,0,4,0,0,0,1,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0
org.apache.tools.ant.AntClassLoader$ResourceEnumeration,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
(more data...)
```

Listing D.2: Property Data - ant-1.8.1 (sample only)

```
Class,DIT,SuperTypes,Interfaces,Type,Classification,EnclosingClass,NLevel,NDIC,NDNC,NDMC,NDLC,NDAC,PubNC,ProNC,PriNC,DefNC,PubNI,ProNI,PriNI,DefNI,PubMC,ProMC,PriMC,DefMC,LocalC,AnonymousC
org.apache.tools.ant.AntClassLoader,1,java.lang.ClassLoader,org.apache.tools.ant.SubBuildListener,C,0,,0,1,0,1,0,0,2,0,0,0,0,0,0,0,0,0,1,0,0,0
org.apache.tools.ant.AntClassLoader$ResourceEnumeration,1,java.lang.Object,java.util.Enumeration,C,2,org.apache.tools.ant.AntClassLoader,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
(more data...)
```

Listing D.3: Inner Class Data - ant-1.8.1 (sample only)