Université de Montréal

# Towards Using Fluctuations in Internal Quality Metrics to Find Design Intents

par

**Thomas Schweizer**

Département d'informatique et de recherche opérationelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures et postdoctorales
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

Orientation Génie logiciel

avril 2020

**Université de Montréal**

Faculté des études supérieures et postdoctorales

Ce mémoire intitulé

# Towards Using Fluctuations in Internal Quality Metrics to Find Design Intents

présenté par

**Thomas Schweizer**

a été évalué par un jury composé des personnes suivantes :

*Marc Feeley*

(président-rapporteur)

*Michalis Famelis*

(directeur de recherche)

*Eugène Syriani*

(membre du jury)

Mémoire accepté le :

*2020-01-10*

# Sommaire

Le contrôle de version est la pierre angulaire des processus de développement de logiciels modernes. Tout en construisant des logiciels de plus en plus complexes, les développeurs doivent comprendre des sous-systèmes de code source qui leur sont peu familier. Alors que la compréhension de la logique d'un code étranger est relativement simple, la compréhension de sa conception et de sa genèse est plus compliquée. Elle n'est souvent possible que par les descriptions des révisions et de la documentation du projet qui sont dispersées et peu fiables – quand elles existent.

Ainsi, les développeurs ont besoin d'une base de référence fiable et pertinente pour comprendre l'historique des projets logiciels. Dans cette thèse, nous faisons les premiers pas vers la compréhension des motifs de changement dans les historiques de révision. Nous étudions les changements prenant place dans les métriques logicielles durant l'évolution d'un projet.

Au travers de multiples études exploratoires, nous réalisons des expériences quantitatives et qualitatives sur plusieurs jeux de données extraits à partir d'un ensemble de 13 projets. Nous extrayons les changements dans les métriques logicielles de chaque commit et construisons un jeu de donnée annoté manuellement comme vérité de base.

Nous avons identifié plusieurs catégories en analysant ces changements. Un motif en particulier nommé "compromis", dans lequel certaines métriques peuvent s'améliorer au détriment d'autres, s'est avéré être un indicateur prometteur de changements liés à la conception – dans certains cas, il laisse également entrevoir une intention de conception consciente de la part des auteurs des changements. Pour démontrer les observations de nos études exploratoires, nous construisons un modèle général pour identifier l'application d'un ensemble bien connu de principes de conception dans de nouveaux projets.

Nos résultats suggèrent que les fluctuations de métriques ont le potentiel d'être des indicateurs pertinents pour gagner des aperçus macroscopiques sur l'évolution de la conception dans l'historique de développement d'un projet.

**Mots-Clés**: Maintenance du logiciel, Conception du logiciel, Historique des versions, Mesure du logiciel, Réusinage

# Summary

Version control is the backbone of the modern software development workflow. While building more and more complex systems, developers have to understand unfamiliar subsystems of source code. Understanding the logic of unfamiliar code is relatively straightforward. However, understanding its design and its genesis is often only possible through scattered and unreliable commit messages and project documentation – when they exist.

Thus, developers need a reliable and relevant baseline to understand the history of software projects. In this thesis, we take the first steps towards understanding change patterns in commit histories. We study the changes in software metrics through the evolution of projects.

Through multiple exploratory studies, we conduct quantitative and qualitative experiments on several datasets extracted from a pool of 13 projects. We mine the changes in software metrics for each commit of the respective projects and manually build oracles to represent ground truth.

We identified several categories by analyzing these changes. One pattern, in particular, dubbed "tradeoffs", where some metrics may improve at the expense of others, proved to be a promising indicator of design-related changes – in some cases, also hinting at a conscious design intent from the authors of the changes. Demonstrating the findings of our exploratory studies, we build a general model to identify the application of a well-known set of design principles in new projects.

Our overall results suggest that metric fluctuations have the potential to be relevant indicators for valuable macroscopic insights about the design evolution in a project's development history.

**Keywords**: Software maintenance, Software design, Version history, Software measurement, Refactoring

# Contents

# List of tables

# List of figures

# Acronyms

**AUC:** Area Under The Curve. 60

**AUROC:** Area Under the Receiver Operating Characteristics. 60

**CBO:** Coupling between objects. 14, 23, 25, 35

**CSV:** Comma Separated Value. 73

**DIP:** Dependency Inversion Principle. xv, 9, 11, 12

**DIT:** Depth of inheritance. 14, 15, 23, 25

**IDE:** Integrated Development Environment. 6

**ISP:** Interface Segregation Principle. 9–11

**LCOM5:** Lack of cohesion of methods 5. 14, 15, 23, 25

**LSP:** Liskov substitution Principle. 9, 10

**NLP:** Natural Language Processing. 75

**OCP:** Open Closed Principle. 9, 10

**OOP:** Oriented Object Programming. 6, 9, 10, 15

**RC:** Refactoring Commit. xiii, xv, 5, 20, 23–30, 34, 39–42, 46–50, 52, 53, 55–57, 59, 60, 62, 64, 71, 72, B-iii

**ROC:** Receiver Operating Characteristics. 60

**RQ:** Research Question. 4, 8, 53

**SRP:** Single Responsability Principle. 9, 10

**VCS:** Version Control System. 2, 3, 7, 12–15, 19, 20, 56, 72, 73

**WMC:** Weighted methods per class. 14, 15, 23, 25

# Acknowledgements

Thank you to my family and my significant other for their support and advice. They were always ready to put things in perspective and help me. I'm very grateful to have them.

I sincerely thanks Doctor Vassilis Zafeiris and Professor Marios Fokaefs for their collaboration and mentoring in this research project. Their experience and insights were invaluable in shaping my work and developing my academic collaboration skills.

I also want to thank the DIRO and ESP for their support through my graduate studies. As an international student, I felt very welcomed and accepted in the university's community.

Finally, I want to express an enormous thank you to Michalis Famelis for being an **outstanding** research advisor, mentor, and person. I'm immensely grateful I had the opportunity to work with him for my Master's degree. He introduced me to the world of research, and I wouldn't have had it any other way.

# Chapter 1

# Introduction

Software development is a complex activity. It requires developers to take into account different perspectives and often switch contexts that are costly to interrupt [**57**]. Since the inception of the field, concerted efforts from research, industry, and hobbyists have been made to improve the development experience by seeking to ease out pains and enable developers to make less mistakes, be more productive, be automatically warned about bugs, improve organization, communicate more efficiently, and ensure that existing features do not deteriorate. Innovations such as automated testing and linting have significantly contributed to not only improve program quality, but also decrease some of the load borne by developers by automating these menial tasks. This arrangement can reap benefits throughout the entire lifecycle of applications: developing is facilitated, enabling developers to create better products for the user and for themselves when they do maintenance down the line. Automated testing ensures that the features of an application do not deteriorate when changes are made to the source code and notify developers when they introduce breaking changes. It is typically implemented with a systematic approach, where developers specify the expected behaviors for each component or subsystem. Automated linting enforce coding conventions and by reformatting the source code accordingly, guaranteeing a common code base between developers that is easier to read.

## 1.1. Modern software development

Nowadays, the software development workflow has evolved to be more agile where a strong emphasis on the creation of working implementation increments is placed in opposition to earlier software development workflows, such as the waterfall model, favoring a detailed up-front design. These models are seldom used nowadays for a reason: their monolithic structures were not able to scale in terms of turnaround and complexity.

On the one hand, this allows flexibility and rapid development cycles, leaving patches, corrections, and enhancements to be applied post-release. On the other hand, this practice tends to accumulate technical debt [**42**] and requires a lot of maintenance effort to continue development, leading to a deterioration in non-functional requirements such as design quality (e.g., maintainability, extendibility, understandability).

Although still an integral part of software development with recognized value, careful and detailed design is often neglected in these iterative workflows and is usually postponed for a later time, often after the system has been released. Given the rapidity of development cycles, this inevitably leads to design erosion [**82**], where assumptions held originally by designers are no longer valid, and design evaporation [**66**], where knowledge about design is lost.

These methodologies have recognized this shortcoming, and they recommend the extensive use of refactoring [**63**] – typically small, local changes that improve design quality without affecting the system's observable behavior – which can be characterized as post-release changes (i.e., maintenance) to the system's structure. They aim to prepare the source code for future extensions and functional enhancements, a process known as "preventive maintenance" [**12**]. However, even if the design can be corrected, refactoring activities may not be explicitly recorded, and changes in design are not always reflected in the available documentation at hand. This can significantly reduce the developers' awareness and knowledge of the system, which in turn can hinder several tasks, including onboarding of new developers and communication between stakeholders.

These contemporary software development practices are enabled by the adoption of Version Control Systems (VCSs). These systems archive code artifacts and integrate themselves into a vast range of development workflows. Their position as a core element of contemporary software development processes is not only due to their low barrier of entry and ease of use. Indeed, their main advantage lies somewhere else. They enable developers to look at and interact with the development history of a project. On top of gaining insights about past changes and reverting them if needed, they bolster the collaboration capabilities of developers, given that software products are often too big and complex for a sole developer, and simultaneous collaborative development is essential.

## 1.2. Problem Statement

In this fast-paced and collaborative environment, developers are called to make changes in any part of the source code which they may not be familiar with, either due to a long absence from this part or that code was modified or added by another collaborator. Thus, to make valuable contributions to the code base, developers must first comprehend the portions of the source code related to their task. They peruse through multiple sources of information, such as code, wikis, discussions, commit messages, and others, often having to sift through potentially out of date information [**66**]. They spend a substantial amount of their time on comprehension to be able to best maintain, add new functionalities, and remove bugs [**46**]. Ultimately, successfully contributing to a project requires contextual design knowledge, which is typically accumulated tacitly overtime throughout the development history of the project.

Unfortunately, extracting relevant and pertinent information about an artifact from its commit history is a daunting task. It involves relying on commit messages and descriptions of the revisions, which – when they exist – can be irrelevant or unhelpful in describing the changes. The only other alternative is to examine what is commonly known as the "diff" – the list of source code changes between commits for each file generated automatically obtained from the VCS – but this process is confusing and slow [**76**] especially since developers are not interested in the comprehensive knowledge offered by the entire revision history of a component: they need particular information about the most relevant changes for their current task or recent changes.

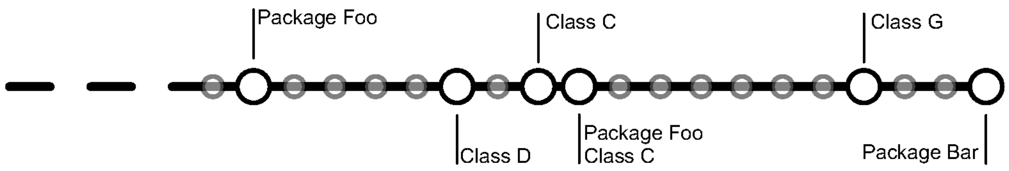


**Fig. 1.1.** Commits containing design changes are highlighted compared to the rest of the version history. Each commit also details which software artifacts had their design changed.

Thus, we aim to find a solution that enables developers to quickly get informed about the design history of the source code that is relevant to their current task. We want to focus on acquiring design information disseminated through the evolution of the code. Current alternatives are lackluster because they are either based on reading documentation, which is often not up to date and thus not a reliable source of information, or they are requiring developers to sift through heaps of code changes for multiple files, which is error-prone and very time-consuming. We sketch the concept in Fig. 1.1 where specific commits relevant to design are highlighted from the other changes registered in the version history.

## 1.3. Research Questions

We define the goal, purpose, quality focus, perspective, and context of our study according to the guidelines defined by Wohlin et al. [**85**]. The *goal* of our study is to analyze metric fluctuations brought by commits for the *purpose* of evaluating the relationship between refactoring, fluctuation of internal quality metrics, and design intent. The *quality focus* is on the effectiveness of employing refactoring and metric fluctuation data for the identification of commits that involve SOLID design principles which are crucial for shaping system design. Study results are interpreted from the *perspective* of researchers with interest in the area of software design and development processes as well as programming practitioners. The results are of interest to developers that need to understand the design of a project through studying important milestones in its evolution and by software architects seeking to review design decisions in committed code in order to confirm their architectural conformance. The *context* of this study comprises change and issue management repositories of a set of open source projects.

We articulate a series of empirical studies with the following Research Questions (RQs):

(1) How does refactoring impact internal quality metrics and design?

(2) Can we classify the changes in internal quality metrics?

(3) What is the effect of environmental conditions to metric fluctuations?

(4) Can fluctuations in metrics be an indicator of the application of SOLID principles?

## 1.4. Approach

"*Design*" is a notion intuitively understood by humans and developers alike. However, the concept englobes several layers and components in the context of software development. While our end goal is to provide information about design changes in general, we have to scope out a reasonable target for this exploratory research. Thus, we choose to use design principles as the embodiment of design changes. Design principles are sets of well-established guidelines used by developers to guide them in the practice of agile design [**55**]. They help identifying and removing design smells that impact the flexibility, reusability and maintainability of software systems.

To achieve this, we first need an intuition about what can characterize these changes in design. We know that refactoring is used to change non-functional aspects of software to facilitate future extensions and reverse design erosion and that we can measure source code to approximate internal qualities of a software [**41**].

Thus, we begin by exploring commits containing refactorings, which we call RCs. Refactoring has been extensively studied for its impact on design quality [**1, 74**], and influence on developer habits with respect to its application on software systems [**80, 70**]. Such studies have focused primarily on the identification of refactorings and refactoring opportunities and on analyzing their impact on design, in terms of the presence of code smells [**29, 79**] or the fluctuation of code metrics [**53**]. The resulting consensus is that refactoring impacts the design of a system in a significant way. This relationship is not incidental: developers purposefully use refactoring to express specific design intentions [**70**] and use recommender systems to identify the most suitable refactorings to best suit their intents [**7**]. Refactoring activity can carry various kinds of design intent, including but not limited to: removal of code smells, resolution of technical debt, the introduction of design patterns, and application of design principles.

We propose to use fluctuations in metrics brought by the code changes from one commit to another to classify different kinds of refactoring activity. *We assume that when a RC affects multiple metrics, improving some while deteriorating others, we have evidence of a developer intentionally changing the design (design intent) by making specific tradeoffs.*

If this assumption holds, we have a good prior to use fluctuations in metrics and their tradeoffs as indicators of design related changes. From there, we have a sound rationale to make an experiment where we try to predict the presence of the application of design principles in commits using metric fluctuations.

To test the assumption, we conduct an initial study on JFreeChart, a library for displaying graphical charts in applications [**26**], using a qualitative analysis. We look at the correlation between a selection of internal quality metrics and the design changes introduced in RCs. Furthermore, we dive deeper into the effect of internal quality metrics changes on different development contexts in a quantitative analysis and we also refine our comprehension of the relation between refactoring and design changes, specifically changes susceptible to embody a design decision, in a third preliminary study. Once the hypothesis is tested, we train a general model on a dataset of annotated commits using supervised learning and evaluate the capacity of the model to recognize the application of the SOLID design principles, a set of design principles commonly used in Oriented Object Programming (OOP), in commits from new projects (projects that were never seen during training).

Our approach is based on existing theories such as internal quality metric measurement [**20**] and quality characteristic appreciation [**6**], and we follow the open source principle of considering the code as the most authoritative source of design information [**24**]. We use a mixed-method approach, consisting of several exploratory case-driven archive analysis, and comparative explorations. We envision our work as complementary to other approaches for extracting tacit and contextual design knowledge such as from discussions [**83**], and commit messages [**18**].

## 1.5. Potential future benefits

In this section, we present the multiple hypothetical benefits of our proposal.

Our proposal could provide developers with an approach that enables them to filter out code changes that are not relevant to design. They would be able to quickly see the relevant last commits affecting the design and understand why it was changed, e.g., a response to the introduction of a new feature or a bug fix. This comprehension allows them to contribute relevant code changes that make use of the existing design efficiently. Moreover, when applied to an Integrated Development Environment (IDE), we could imagine a dedicated pane that would provide developers an overview of the moments in time when the design changed for the software artifact (e.g., a file, a class, a model) they are currently editing, helping the developer in his exploration and comprehension of the artifact, and allowing him to quickly go back in time or get more information if he interacts with one of the moments in time displayed.

This approach could also be applied to the code reviewing process where developers are assigned changes made by other developers to review [**52**]. This practice is a staple in major software companies [**52**]. It allows to catch bugs early and fosters a shared knowledge of the system among its developers. By integrating our approach into the review process, we could detect when a set of changes is susceptible to contain design changes and warn the developers that an increased attention should be given for these code changes. Reviewers would be able to see the modifications to the design and conduct sanity checks to make sure that changing the design is the best solution. Moreover, this application provides a good opportunity to include a feedback mechanism that would help our system to improve its predictions as reviewers use it.

Another benefit would take place during the onboarding process where new developers get acquainted with the software systems. Our approach would help them identify key moments in the construction of the software, informing them about the motivations behind the present architecture and state of software which they wouldn't have otherwise with their minimal experience.

Ultimately, we want to generate a descriptive, history-based meta data for each code artifact that could be used by developer and other stakeholders to synthesize documentation, and to empirically assess software and software projects and make informed decision. With this research, we take the first steps towards this direction.

## 1.6. Contributions

We make the following contributions:

(1) A systematic methodology to mine internal quality metric fluctuations from VCS.
(2) An open source toolchain that implements it, called MetricHistory [**78**].
(3) A deep qualitative analysis of the revisions containing refactorings in one open source project, JFreeChart.
(4) A quantitative study of the metric fluctuations in 13 open source projects.
(5) A scheme for classifying metric fluctuations.

(6) A public data repository of historical internal quality metric fluctuations.

(7) A manually annotated dataset composed of 928 commits identifying the presence of SOLID principles.

(8) A procedure to build and train a classifier to detect SOLID principles in version histories and its empirical validation.

(9) A general model that can identify the application of the SOLID principles in commits.

(10) The outline of an approach for filtering a project's revision history to a set of revisions that have a high likelihood to carry design intent.

## 1.7. Structure of the thesis

The thesis is organized as follows. Chapter 2 explains the core background concepts. In Chapter 3 we illustrate the overall design of the studies as well as the processes used to gather the various data used throughout our research. Then, in Chapter 4 and Chapter 5 we describe the studies performed to test our assumption and answer the RQs. We present the related work in Chapter 6 and summarize our findings and discuss future work in Chapter 7.

## 1.8. Work attribution

The research presented in this thesis is an international study project with Prof. Michalis Famelis, Prof. Marios Fokaefs and Dr. Vassilis Zafeiris. Particularly, Dr. Zafeiris collaborated with me in building the annotated datasets (Section 4.1, 4.3, 5.1), distributing mining workload across different computers (Chapter 3), analyzing the effect of releases on metric tradeoffs (Section 4.2.2.3).

# Chapter 2

# Background

## 2.1. Software Design

Software design is the process following the requirements collection and preceding the implementation of a system or its components [**33**]. The purpose of this activity is to create a structure of the program, similarly to an architect drawing the plans for a new building, that satisfies the requirements established previously. However, the designer also has to balance other considerations such as extensibility, modularity, and maintainability that will affect the software's quality in the long term. Balancing the expectations of multiples stakeholders is difficult and it is impossible to satisfy everybody. To mitigate the risks and focus on the right design areas, developers use design principles such as SOLID principles in OOP and design patterns as heuristics to guide them [**9, 55**].

## 2.2. SOLID Principles

The SOLID principles are a set of well known design principles in the oriented object community. It is composed of five principles: Single Responsability Principle (SRP), Open Closed Principle (OCP), Liskov substitution Principle (LSP), Interface Segregation Principle (ISP), DIP [**54**]. The SOLID principles are meant to guide the practice of agile design by helping to identify and remove design smells that impact the flexibility, reusability and maintainability of system design. In theory, the implementation of decisions based on SOLID principles is performed with manual or automated refactorings.

SRP specifies that a software artifact such as a class, method or package should only have one responsibility, or in other words, only one reason to change. This principle encourages the separation of concerns which contributes to increasing the cohesion of artifacts and decreasing their coupling with other elements. For example, if a class handles the creation and logging in of users, an application of SRP could lead to split the class in two. One class would now have the responsibility of creating users and the other one would have the responsibility of logging them into the system. This way, if the creation procedure changes, the class handling logins will be left untouched, reducing the chance to induce bugs and facilitating the work of developer.

OCP, credited to Bertrand Meyer [**58**], specifies that software artifacts should be open for extension but closed for modification. The idea is to reuse existing components by extending them to new needs rather than modifying them directly. The advantage of this concept is to avoid problems stemming from modifying artifacts that are used by multiple components inside a system. If you modify the original artifact it will change the behaviors of the dependents in unforeseeable ways which often lead to regression problems and backward incompatibilities which are hard to solve. By extending the artifact's features, you do not touch the original behavior. A very popular example is the use of polymorphism in object oriented languages such as Java. The developers can change the behavior of a base class by implementing a subclass. For example, you can extend a class representing a collection of heterogeneous elements through a subclass to implement the concept of a mathematical set. The newly created subclass is still representing a collection but it is now enforcing particular semantics.

LSP is a concept introduced by Barbara Liskov [**50**]. In the context of type theory in OOP, it specifies, that for a type $T$ and a subtype $S$, the objects of type $T$ can be replaced by objects of type $S$ without changing the behavior of the systems where $T$ was expected. This principle can be applied to several situations in practice such as mixed instances cohesion problems [**64**]. This problem arises when a class's feature is provided in at least a couple of implementations in its instances. The consequence is that the instances will have some attributes or methods that are undefined or have unexpected secondary effects. By applying LSP, the class will be transformed into a base class $T$ and each implementation is sent to a subtype $S$, ensuring a consistent behaviour for the feature provided by the class.

ISP specifies that is it better to have multiple interfaces, each dedicated to one aspect of a feature or concept, rather than one big interface between two clients. This principle also promotes a separation of concerns and aims to improve cohesion. As an example, we will model a modern printer. Modern printers have multiple capabilities such as scanning, printing, and sometimes faxing. It can be tempting to create one interface regrouping all these functionalities as an "*All in one Printer*" and implementing our "*Modern Printer*" after it. Now, you reuse the interface to implement a simple machine that only prints. It leaves you with two features from the interface your "*Simple Printer*" can't provide and you have to code work-arounds to handle these impossible cases. By using ISP, you can handle this situation elegantly by separating the *All in one Printer* interface in three: "*Printer*", "*Scanner*", "*Fax*" and adapting *Modern Printer* and *Simple Printer* in consequence. The former will implement all three interfaces while the latter will only need to implement *printer*.

DIP encourages developers to build components in such a way that they depend on abstractions rather than implementations. More specifically, high level components should not depend on lower level components. The consequence entails that abstractions will not depend on details, guaranteeing the generalizability, but details depend on abstractions, therefore inverting the traditional dependency relationship where the main component of a system would depend on detailed components. Developers usually implement this principle by adding a layer of abstraction between dependencies. For example, in a system where the class *Printer* depends on a class "*Paper Tray*" as shown in Fig. 2.1 a), we would abstract *Paper Tray* behind an interface such as "*Paper Supplier*". This way, the higher level component, *Printer*, would only about *Paper Supplier* and not the lower level component *Paper Tray* as illustrated in Fig 2.1 b). When the components are grouped in different modules, it is possible to push the principle even further by reorganizing the location of the abstraction. In our example, *Printer* is in a module "*machine*" and *Paper Tray*, as well as *Paper Supplier* are in a module "*supplies*". This is quite a natural way to group elements as they belong the same concept. However, this layout introduces a dependency between the *machine* and *supplies* because *Printer* depends on *Paper Supplier*, which contradict DIP as we can consider that the former still represents a higher level than the former. We solve this problem by moving the *Paper Supplier* interface to the module *machine* as shown in Fig. 2.1 c). Now dependencies are fully inversed as the lower level component, *Paper Tray*, depends on a the higher level concept, *Paper Supplier* as it is now close to the class *Printer* and also its only dependency.

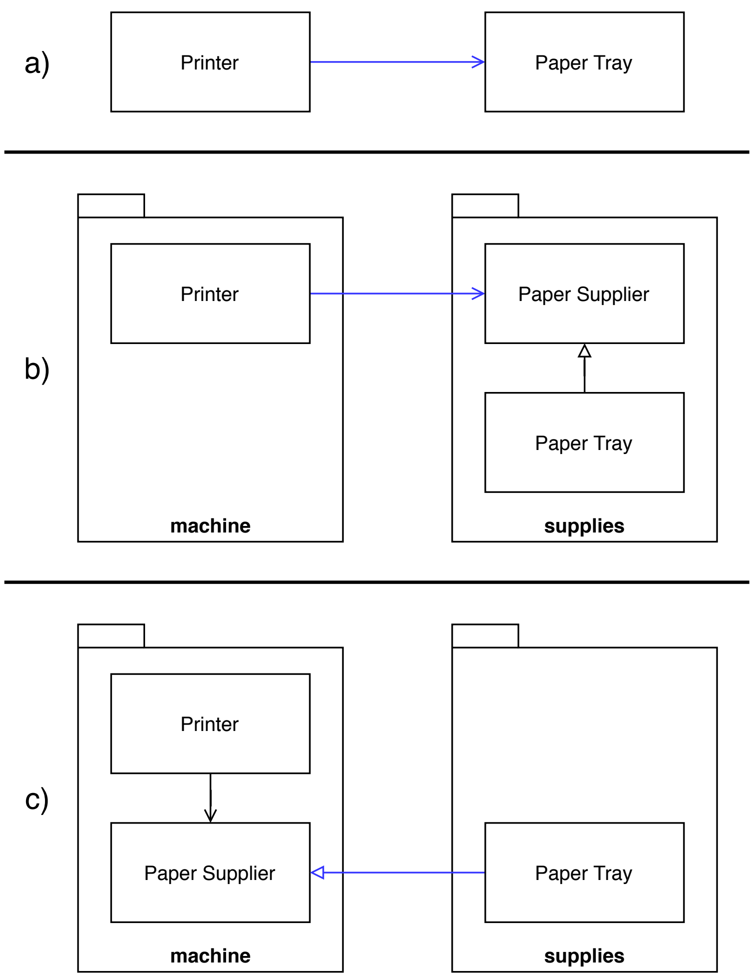


**Fig. 2.1.** Illustration of the application of DIP on a toy example. The colored arrow represents the dependency between a high component (left) and a lower level component (right). Note how the arrow changes direction from layout b) to c).

## 2.3. Version Control Systems

VCS are software systems that record changes to a set of files over time in an automated fashion. Originally, most developers would copy and date the files they wanted to "save" into another folder on their computer. However, this technique doesn't scale well and is not practical when collaborating with other developers. VCS solves these issues by providing developers an easy way to store changes and share their code with other colleagues that can also propose changes into a ledger, commonly referred to as the "*changelog*", where all changes are recorded. Developers also gain the capacity to go back to any change they made and branch out from there, building an alternate version of the software.

Moreover, a software project under version control is referred to as a "*repository*". A new entry to the changelog, in the form of a set of changes in the source files, is called a "*commit*" or a "*revision*" depending on the terminology used by each VCS. A "*tangled*" commit denotes instances where multiple unrelated sets of changes are cohabiting in the same commit, although the changes could be partitioned in different commits [**32, 72**]. On the opposite, clean or tangle-free commits, have only one set of auto-contained, cohesive changes.

Some VCSs provide distributed versioning, enabling developers to commit changes locally on their system and then share their changes to a remote repository accessible by others. Examples of known and popular VCSs include *Git* [**16**], *Subversion* [**21**], and *Mercurial* [**51**].

## 2.4. Mining Software Repositories

Mining of Software Repositories is a method of archival research where researchers look at the evolution of the artifacts in the project through time [**15**]. The adoption of Open Source and VCS enables researchers in software engineering to study the evolution of a software project in terms of specifications, source code, social interactions between developers, and bugs from its inception to present days. To study these, they use the process of mining, which is the systematic extraction of information relevant to the subject of study in the version history of a project [**39**]. For example, it is common to examine the changes in files or the VCS meta-data. Recently, the community has done substantial work on software defects and studying the dynamics of Open Source collaboration [**19**].

Historically, this technique is based on early ideas of Ball et al. [**5**], where they make the argument that VCSs contains a lot of useful contextual information by exploring the evolution of class relationships through time. Shortly after, an approach was implemented by leveraging the changes in between versions characterized by software releases with ad-hoc toolchains to detect logical coupling [**25**]. Then, later studies introduced experiments using a smaller granularity of versions such as commit to commit. The early work of Zimmerman et al. is an example where they leveraged VCS information to detect files commonly changed together and propose recommendations [**87**]. More recently, initiatives such as GitHubTorrent [**28**] democratized this practice by offering convenient facilities for researchers to select and retrieve massive amounts of VCS meta-data from version control systems.

Compared to them, we focus our efforts on extracting software metrics and their fluctuations instead of analyzing the VCS meta-data or file-based changes. Another difference is that our approach works at scale with the combination of our versatile tool and its integration to a distributed computing system with Akka, a framework to build powerful reactive, concurrent and distributed applications more easily [**47**].

However, software repository mining is not an approach suitable for every type of study. Especially when working with public repositories, which are known to be often personal projects, or sometimes are not even related to software development but used as a data storage service. Moreover, even legitimate software projects can be problematic since their development workflows are sometimes coupled with private tools such as bug trackers or project management systems. These tools contain valuable information about the development and history of a project, but they are often inaccessible to repository miners. Thus, when mining software repositories, researchers have to be careful about the provenance and the relevance of the repositories, and their data with respect to the objective of their research[**40, 17**].

## 2.5. Metrics

We compute the metrics to measure the changes in the projects using SourceMeter, a static analyzer that supports a wide variety of metrics and granularities (e.g., method, class, package) [**20**]. For this study, we focus on class metrics. SourceMeter offers 52 metrics for classes, distributed in 6 categories: *cohesion metrics (1), complexity metrics (3), coupling metrics (5), documentation metrics (8), inheritance metrics (5), size metrics (30)* with the respective number of metrics for each category. The detailed documentation for each metric and how it's calculated is available on SourceMeter's publisher website [**3**].

In Chapter 4, we use a subset consisting of four metrics: Coupling between objects (CBO) to measure coupling, Depth of inheritance (DIT) for inheritance complexity, Lack of cohesion of methods 5 (LCOM5) for cohesion, and Weighted methods per class (WMC) for method complexity. These metrics were selected in a process explained in the aforementioned chapter.

CBO measures coupling between objects by counting the number of classes a class is depending on. A high count indicates that the class is linked to many others which can cause maintainability problems such as a brittle reliability when the dependencies change often.

DIT measures inheritance complexity by counting recursively the number of subclasses a class has. In other words, it measures the length of the path to the deepest ancestor. The deeper the inheritance tree runs, the harder it is to modify or reuse the elements of the hierarchy because they become very specialized and depend on their parents for functionalities. LCOM5 measures class cohesiveness by counting the number of methods linked to the same attribute or abstract method as OOP practices advise that a class should only have one responsibility. If multiple methods do not share the same attributes or abstract methods, it creates groups of methods that are indicators of a fragmented class. The higher the lack of cohesion is, the lowest the responsibility of the class is crisply defined, often resulting in brittle implementations of features that do not actually belong together. Finally WMC, measures the complexity, in terms of independent control flow path, for a class. It is calculated by summing up the McCabe's Cyclomatic Complexity [**56**] for each method and the `init` block of a class. The higher the number, the harder it is to understand, and thus verify and maintain, the class and its methods because of the number of different control flow paths available.

## 2.6. Summary

In this section, we introduced the core concepts we rely on in our research. Software design represents the end phenomenon we are trying to capture and recover. However, the notion of software design, while well-defined, can be somewhat abstract in practice. Thus, we use the SOLID design principles as a concrete and measurable manifestation of software design concerns. In addition, we rely on VCSs as the framework that allow us to retrieve the evolution information of a software project as its history of change is recorded through it in the form of commits. Our study is guided by the principles and methodologies developed for research in the area of Mining Software Repositories. Finally, we introduce the metrics we use to characterize the source code changes happening between versions.

# Chapter 3

# Data collection

In this chapter, we present our overall study design. We describe our processes and the different datasets created. We specify the scope, the population and its sampling, and the data collection techniques. Our approach consists of six steps: A) Commit selection, B) Refactoring identification, C) Mining metrics D) Converting metrics' format, E) Computing changes, and F) Aggregating metrics. We detail them in the following sections. Fig. 3.1 provides an overview of our approach and datasets.

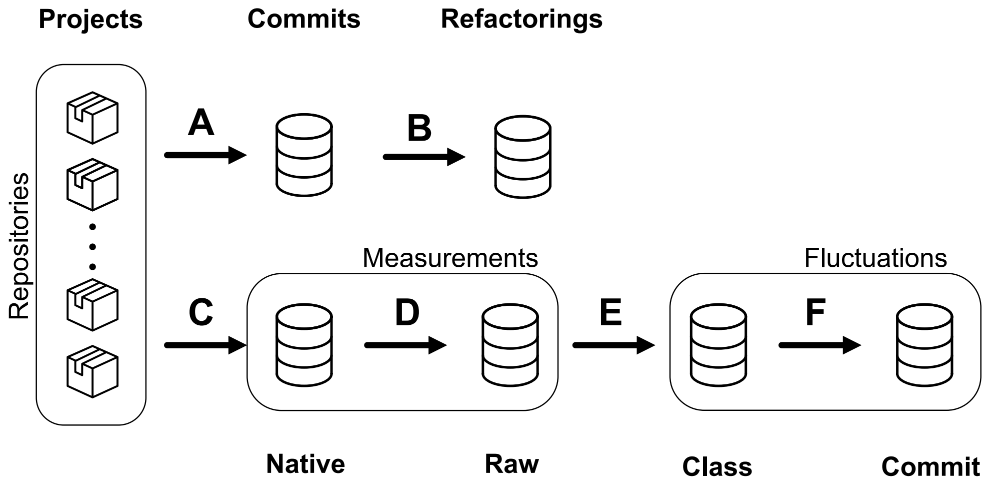


**Fig. 3.1.** Different datasets mined from the source code of the project

## 3.1. Design

### 3.1.1. Unit of analysis

Our unit of analysis is the *commit* or *revision*. In version control systems, a revision represents the list of differences for the files that changed in the project from a previous – also known as parent – version to the next version. Revisions are groups of incremental changes between a pair of versions, each revision represents an atomic unit of work, usually containing a cohesive set of changes although it is not always the case in practice [**2**]. We can think of the list of revisions of a project as its evolutionary history; each revision embodies a version of the software.

### 3.1.2. Target population

We aim to understand the patterns of metric fluctuations in software projects written in Java under version control. The target population of our study is therefore the set of all revisions present in the version histories of such projects. The version history contains the incremental changes applied to the project until its latest state of development as explained in the previous paragraph.

### 3.1.3. Sampling technique

We selected revisions from the version history of a sample of 13 popular open-source Java projects. The selection was conducted using a mix of *convenience*, *maximum variation*, and *critical* sampling based on a blend of several attributes such as projects' popularity amongst developers, usage as research subjects, size, number of contributors, platform, development style, and type (e.g., library, desktop application). Each project has a website and a public repository of source code under an arbitrary version control system – The type of the VCS was not a criterion.

## 3.2. Project selection

We selected 13 popular open-source Java projects. The projects are listed in Tab. 3.1 with the branch, source code location, number of commits mined and size. The projects also came with optional issue trackers, mailing lists, forums, or changelogs (i.e., files maintained by developers to keep track of changes happening in commits). For projects that use Subversion

**Tab. 3.1.** List of the software projects retained.

| Project | Source code | Branch | Commits | Size (SLOC) |
|---|---|---|---|---|
| Ant | `https://github.com/apache/ant` | master | 14 234 | 139k |
| Apache Xerces-J | `https://github.com/apache/xerces2-j` | trunk | 5 508 | 142k |
| ArgoUML | `http://argouml.tigris.org/source/browse/argouml/trunk/src/` | trunk | 17 797 | 176k |
| Dagger2 | `https://github.com/google/dagger` | master | 1 969 | 74k |
| Hibernate ORM | `https://github.com/hibernate/hibernate-orm` | master | 9 320 | 724k |
| jEdit | `https://sourceforge.net/p/jedit/svn/HEAD/tree/jEdit/trunk/` | trunk | 22 873 | 124k |
| Jena | `https://github.com/apache/jena` | master | 7 112 | 515k |
| JFreeChart | `https://github.com/jfree/jfreechart` | master | 3 640 | 132k |
| JMeter | `https://github.com/apache/jmeter` | trunk | 15 898 | 133k |
| JUnit4 | `https://github.com/junit-team/junit4` | master | 1 972 | 30k |
| OkHttp | `https://github.com/square/okhttp` | master | 1 951 | 61k |
| Retrofit | `https://github.com/square/retrofit` | master | 1 038 | 20k |
| RxJava | `https://github.com/ReactiveX/RxJava` | 2.x | 4 137 | 276k |

(jEdit, ArgoUML) as their VCS, we first migrated their version history to Git using GitHub Importer [**27**].

## 3.3. Commit selection (A)

After downloading the projects onto disk, we extracted the commits from the default branch of each of the 13 projects until 2018-12-31 (included). A branch represents an independent line of development [**11**]. We focus on the default branch because it is often this rendition of the development that will be released to the public. It represents the mature code changes of a project. From this set, we exclude commits that are the results of a *merge operation*. In VCS, “merging” means to integrate all the changes from a “source” branch to a “destination” branch in a single commit. The resulting commit is generally hard to read for humans and presents little interest for analysis as the individual changes can be found on the source branch.

We used the following command to obtain the list of commits to be analyzed for each project (where `<branch>` is the name of the default branch):

```
git log <branch> --pretty="%H" --no-merges --until="2018-12-31"
```

Overall, the 13 projects represent a cumulative 107 449 versions of projects spanning 20 years of software development history.

## 3.4. Refactorings (B)

As explained in the introduction, our study is interested in the refactoring activity of developers. To isolate the revisions containing refactorings, we used RMiner [**81**], a specialized tool that can detect refactorings automatically in the history of a project with high recall and precision compared to other similar tools, such as RefDiff [**69**]. We ignore refactorings related to tests because we are not studying the role of tests in a software's design. The refactoring detection yields a list of revisions containing at least one refactoring. Henceforth, we refer to these revisions as RC. Each of these RC is also accompanied by a detailed list of the refactorings it contains.

## 3.5. Native dataset (C)

We used *SourceMeter* [**20**] to calculate the software metrics for individual revisions. We automated the execution of SourceMeter to run in batch multiple commits of a project with our command line tool MetricHistory [**78**].

Calculating the metrics for thousands of revisions is a computationally intensive task, especially for large projects (e.g. jEdit, Hibernate ORM). It can be seen as the equivalent of calculating the metrics for 107 449 projects. To accommodate this process, we built a distributed computation system based on the Akka toolkit and runtime [**47**]. Each computation node invokes MetricHistory tasks to calculate the metrics for individual revisions of a project. Job assignment is performed by a scheduler node that also collects and stores the results in a data repository.

MetricHistory is an extensible tool designed to collect and process software measurements across multiple versions of a code base. The measurement itself is modular and executed by a third party analyzer, in the case of this study we use SourceMeter. The core design principles of MetricHistory is to integrate into any toolchain using its command line interface or Java API. It also aims to be easily customizable by its modular architecture. For example, adding a new analyzer or supporting another VCS only requires to implement an interface. MetricHistory is also operationally modular to accommodate the explorative workflows of researchers. The results of each step mentioned in this chapter can be saved or recomputed at any moment so one can resume from any point in their workflow and interchange data between steps. In the case of the aforementioned distributed setting, each worker goes through all the steps at once for one commit only. However, if you have a computer powerful enough or a smaller number of projects, it also support to calculate all the measurements for one step for all commits in batch manner and save the results for a later use, or analysis through a custom toolchain. A detailed description of MetricHistory's features, installation, and usage is presented in Appendix A.

## 3.6. Raw dataset (D)

Since MetricHistory can make use of different analyzers – each with their own output format – we make a distinction between the format of the metrics used to calculate the fluctuations and the output format of our analysis tool. Supporting this transient step make our toolkit modular, allowing future use of, not only different analyzers, but also pre-compiled measurements from other researchers and studies.

Using MetricHistory again, we convert the class measurements into a common basic "RAW" format. This scalable file format identifies the measurements for a given artifact through multiple versions. This dataset contains the metrics generated by SourceMeter for each class (identified by its canonical name i.e. *com.example.FooBar*) for each version obtained in step **A**. You will find below the 5 first lines of typical raw file. Only three metrics are shown for readability.

```
commit;class;LOC;CBO;DIT
9b23cc2184438f93923c972c45b7caeb43d77d24;org.animals.CowRenamed;19;2;1
9b23cc2184438f93923c972c45b7caeb43d77d24;org.animals.Dog;13;1;2
9b23cc2184438f93923c972c45b7caeb43d77d24;org.animals.Labrador;124;0;1
9b23cc2184438f93923c972c45b7caeb43d77d24;org.animals.Poodle;24;5;1
```

For projects with a large number of commits to analyze, we can separate the results by commit. In other words, we create one file for each commit instead of writing all the results into a single file for all commits and classes.

## 3.7. Fluctuations dataset (E)

Using the transformed data acquired in **D**, we use MetricHistory to compute the change of metric for each class from each commit in **A**. For example, if class *Foo*'s **LOC** metric is measured as 3 in commit 1 and is measured at 5 in commit 2, the change of metric *LOC* for *Foo* in version 2 is of $5 - 3 = +2$. Commit 1 is the set of code changes directly before commit 2, also referred as the "parent" commit.

## 3.8. Aggregation (F)

The last step converts the metric fluctuations for each class (class fluctuations) into metric fluctuations describing the changes in a commit (commit fluctuation). We aggregate the fluctuations of each metric across every changed class into one value per metric using a naïve summation. For example, if commit 2 has classes *Foo* and *Bar* that have changed with the metric fluctuations for metric $LOC$ as +20 and +30 respectively. Then the metric fluctuation at the commit level will be $20 + 30 = +50$.

We note two crucial details. First, we only retain the metric fluctuations from classes that have changed; we ignore classes that were added or deleted in order to capture design changes inside classes and not inside a package or group of components. Second, we count the number of metrics that are affected by a change during the aggregation process. This prevents metrics changes to disappear at the commit level (the sum is 0) when metric fluctuations at the class level are cancelling each other.

## 3.9. Summary

In this Chapter, we presented our pipeline and the datasets collected. Once the projects are identified, the first steps are to determine the commits to include for each project (A) and then find the commits in this selection that contain refactorings (B). Then, using MetricHistory, we extract the metrics at each commit for all projects to create the native dataset (C). After that, we use MetricHistory to transform the native dataset in the raw dataset that contains the metrics for each class across all commits (D). At that time, we create the fluctuation dataset that contains the metric fluctuations for each class for all commits (E). Finally, we aggregate the class fluctuations with a summation operator to create the metric fluctuations to represent individual commits (F).

# Chapter 4

## Exploratory studies

In this section, we are interested to lay the basis of knowledge regarding metric fluctuations and determine if our hypothesis presented in Chapter 1 is sound. We hypothesized that when a RC affects multiple metrics, improving some while deteriorating others, we have evidence of a developer intentionally changing the design (design intent) by making specific tradeoffs. Specifically, we explore the concept of metric fluctuations and their relation to development activities in the context of design. We conduct three experiments from the datasets created through the approach presented in Chapter 3.

The first experiment is a qualitative study on a subset of commits of JFreeChart where we classify revisions into four categories based on the pattern of the metric fluctuations. We focus on RCs because they are likely to contain intentional structural changes from developers. We then manually analyze each commit to understand its changes and their relation to the categories. In the second experiment, we conduct a quantitative study where we analyze all the commits of the 13 projects under different lenses to test the generalizability of metric fluctuations. Finally, in a third experiment, we look at the capacity of metric fluctuations to act as indicators for design related changes, specifically design intent, in a sample of commits.

In the subsequent explorations, to detect and characterize changes in the software's design, we measure its internal quality properties using multiple metrics. We focus our analysis on four metrics on the interval scale: CBO to measure coupling, DIT for inheritance complexity, LCOM5 for cohesion, and WMC for method complexity. We focus on this small set of metrics as they are considered some of the most representative for their particular properties and good indicators of design quality [**53, 74**]. In addition, to locate intentional changes, we scope our exploration using refactorings as they are embodying conscious changes made by developers [70]. Finally, when coupled with our aforementioned hypothesis, it means that commits containing tradeoffs expressed by metric fluctuations have a strong likelihood to be indicators of the implementation of design decisions made by the developers.

By characterizing the metric fluctuations across the history of development, we aim to provide a baseline for understanding changes to identify development activities, and facilitate the detection of specific change patterns that can be relevant as documentation to developers or other stakeholders.

## 4.1. Exploring JFreeChart

In this section, we present an exploratory study to estimate RCs' contributions to design quality through the use of internal quality metrics. We studied the development history and refactoring activity of JFreeChart [1], a well-studied [1, 86] software project in order to answer RQ1.

### 4.1.1. Setup

4.1.1.1. *Project selection*

We selected the open-source project JFreeChart. It provides a chart library written in Java which enables developers to integrate professional-looking charts in their programs. This project has been studied extensively by the refactoring community [1, 86]. Its medium size (∼600 classes) and history (over 10 years old) is ideal. It is big enough to be relevant in quantitative analysis, while being small enough to allow manual and qualitative analysis. Its size may also support relatively strong conclusions and help to guide our future studies. The project has been used by a variety of applications from different domains over the years and is still actively developed[2].

4.1.1.2. *Objects*

The JFreeChart project is composed of two source code repositories, two bug trackers, mailing lists, a forum, and a website:

*Repositories*

[1]`http:/www.jfree.org/jfreechart/`

[2]`http://www.jfree.org/jfreechart/users.html`

- `https://sourceforge.net/p/jfreechart`
- `https://github.com/jfree/jfreechart`

*Bug trackers*

- `https://sourceforge.net/p/jfreechart/bugs`
- `https://github.com/jfree/jfreechart/issues`

**Mailing lists:** : `https://sourceforge.net/p/jfreechart/mailman`

**Forum:** : `http://www.jfree.org/forum/index.phps`

**Website:** : `http://www.jfree.org/jfreechart/`

This project has a particularity: The development started on SourceForge and was then imported to GitHub. However, the content of the bug tracker was not imported to GitHub at once; they gradually stopped using the one provided by SourceForge and moved gradually to the one provided by GitHub. As a result, the source code and the issue repositories are split between the two platforms.

Thus, we selected all the revisions available on the GitHub repository before 2018-05-01. This selection contains 3 646 revisions covering over 10 years of development in a mature project. Each revision is characterized by its source code, comments, commit message, and updates to the changelog (this artifact is edited by the developers to detail the modifications to the source code for every revision; it is stored in the same repository as the source code).

### 4.1.2. Computing the dataset

Using the procedure described in Chapter 3, we calculated the metric fluctuations at the commit level for all commits in the GitHub repository before 2018-05-01 for the metrics CBO, DIT, LCOM5, and WMC. These quadruples are further used to proxy the direction of change in internal design quality. The focus is on the direction of change that constitutes a trend, rather than on change magnitude. To better understand such trends, we defined four intuitive scenarios to classify the patterns of activity for a RC, given the metric changes. These scenarios are described in the next paragraphs and summarized in Tab. 4.1.

Scenario 1: RCs with no change in metrics. An example of this scenario is a revision where a refactoring was found to have been applied, but no change in any of the selected metrics was found. This is the case for refactorings like renames. Based on the metrics we have selected,

**Tab. 4.1.** Summary of the scenarios

| Scenario | Definition |
|---|---|
| 1 | No metric changes |
| 2 | One metric changes |
| 3 | At least two metrics change, and the changes are in the same direction (all improve or all worsen) |
| 4 | At least two metrics change, and the changes are in mixed directions (some improve and some worsen) |

Scenario 1 instances are not normally expected to represent important design decisions, but rather pure functionality addition or understandability enhancements.
Scenario 2: RCs with a change in a single metric. In this scenario, we include RCs that affect a single metric, positively or negatively. Especially, in the case of positive impact, these instances could correspond to targeted changes to specifically improve the particular metric. While this shows clear intent, the intent is not necessarily related to design decisions.
Scenario 3: RCs where all metrics change monotonically towards improving or declining direction. This scenario includes RCs where more than one metric was impacted. A special inclusion condition is that all the affected metrics should have changed towards the same direction, either all positively or all negatively. Similar to Scenario 2, RCs in this scenario show clear intent. However, due to the scale of change and the impact on metrics, the intent is more inclined to be closer to a design decision.
Scenario 4: RCs where multiple metrics change in different directions. Scenario 4 is the same as Scenario 3 in terms of multiple metrics being affected, with the important difference that not all metrics change towards the same direction. One popular example is the metrics for cohesion and coupling, which in many cases change at the same time, but in opposite directions, especially during remodularization tasks [**74**]. In our view, these instances are the most interesting ones, as they indicate conflicting goals.

In the context of our work, we call instances of Scenario 4, "*design tradeoffs*". Indeed, we established earlier that changes in internal quality metrics in opposite directions translate to tradeoffs in internal quality, which are likely to capture design tradeoffs.. In practice, a design tradeoff is a situation where a change, i.e., a refactoring action, would result in a controversial impact to design quality; while some dimensions are improved, others may deteriorate. In this situation, the developer will have to make a decision as to which metrics and quality aspects are more important than others (given the current requirements) and eventually settle for specific tradeoffs. This is why we consider instances in Scenario 4 to be closely related to design decisions. It is also possible that design decisions also appear in instances of Scenario 3, where there is no tradeoffs but a clear direction of changes measured.

### 4.1.3. Manual Evaluation

We manually analyzed each RC to identify the design intent behind applied refactorings. We based our analysis on code and comment inspection, commit messages, and the changelog of refactored classes. Specifically, we studied the developers' design intent from two perspectives:

(1) The involvement of design decisions in the refactoring process, i.e., whether the developer applied the identified refactorings as part of introducing new design decisions or enforcing design decisions that were established in previous revisions.

(2) The type of implementation task the developer was engaged in, while changing code structure through refactoring, i.e., whether any design decisions were enforced as part of (a) refactoring low quality code, (b) implementing new features, or (c) fixing bugs.

The detection of design decisions in RCs is a rather challenging task since it requires understanding not only the changed code parts, but the overall design of affected classes. Moreover, determining whether a set of refactorings enforce a past design decision, requires tracing back to previous revisions of refactored code. A successful strategy to improve this process was to begin the analysis with the oldest refactoring and then go forward in time: This helps the reviewer to understand the evolution of the design. Additionally, the developers of JFreeChart scrupulously maintain a changelog of their source code changes at the project and file level, giving us insights about their intents.

In order to reduce the subjectivity of this process, the evaluation was performed independently by two of the authors and it was followed by a strict conflict resolution procedure. The inter-rater agreement between their assessments was initially moderate, indicated by a value of 0.49 for Cohen's Kappa [**14, 43**].

### 4.1.4. Results

We have automatically analyzed 3 646 commits in the version history of JFreeChart with an extended version of RMiner [**81**]. The tool identified 247 refactoring operations in the production code that were distributed across 68 revisions. The automatically identified

**Tab. 4.2.** Refactoring operations in JFreeChart

| Refactoring Type | Count |
|---|---:|
| Extract And Move Method | 6 (2.5%) |
| Extract Method | 80 (33.5%) |
| Extract Superclass | 2 ( 0.8%) |
| Inline Method | 4 ( 1.7%) |
| Move Class | 20 ( 8.4%) |
| Move Method | 6 ( 2.5%) |
| Move Source Folder | 1 ( 0.4%) |
| Pull Up Attribute | 12 ( 5.0%) |
| Pull Up Method | 14 ( 5.9%) |
| Rename Class | 15 ( 6.3%) |
| Rename Method | 79 (33.0%) |
| **Total** | 239 (100%) |

refactorings were manually validated and eight of them (7 cases of EXTRACT METHOD, 1 case of RENAME METHOD) were rejected as false positives. The *refactoring revisions* containing them did not include any true positives and were also rejected from further analysis (4 revisions). Tab. 4.2 presents the distribution of true positives to different refactoring types in the 64 remaining RCs. The 64 RCs were further processed in order to measure the differences of internal metrics for all changed classes, as explained in Section 4.1.2.

We then automatically classified each RC to one of the four scenarios introduced in Section 4.1.2. We show the classification in Tab. 4.3. Noticeably, a large part of RCs (29.7%) do not involve changes to internal metrics (Scenario 1). Source code changes in these revisions are due to rename and move class refactoring operations. RCs with a single changed metric (Scenario 2), amount for 35.9% of total revisions. These revisions involve mainly extract method refactorings that affect the WMC metric. Revisions classified to Scenario 3 make up 25% of the total. In them, developers applied a more extensive set of refactoring operations, such as MOVE ATTRIBUTE/METHOD, EXTRACT SUPERCLASS, and MOVE CLASS. Such refactorings have a combined effect on internal metrics, either improving or deteriorating all of them. Finally, we found that in 9.4% of RCs multiple metrics are changed towards

**Tab. 4.3.** RCs for each Scenario

| Scenario | Revisions (%) |
|---|---|
| 1 | 19 (29.7%) |
| 2 | 23 (35.9%) |
| 3 | 16 (25.0%) |
| 4 | 6 ( 9.4%) |

**Tab. 4.4.** Implementation tasks and RCs

| Task type | Revisions (%) |
|---|---|
| Refactoring | 30 (46.9%) |
| Feature Implementation | 29 (45.3%) |
| Bug Fix | 5 ( 7.8%) |

different directions. Such revisions usually involve *design tradeoffs*, i.e., improvement of a design property of one or more classes at the expense of deteriorating another. For instance, a Move Method refactoring may improve the cohesion of the origin class at the expense of increasing the coupling of the destination class.

We summarize the types of implementation tasks that developers were involved in RCs in Tab. 4.4. We determined the type of implementation task through inspection of code differences combined with analysis of commit logs, and embedded change logs of refactored classes. In several cases, commit and change logs included references to issue tracking identifiers. Revisions with a pure refactoring purpose (termed "root canal" by [**61**]) correspond to 46.9% of total revisions. Most of these revisions (20 out of 30) involved only renaming operations, while the rest applied Extract/Inline/Move Method refactorings. Simple refactorings (Extract/Move Method) are also applied within revisions that focus on fixing bugs. The most complex and, also, interesting cases of refactorings are part of revisions that focus on new feature implementation tasks (termed "flossing" by [**61**]). These revisions correspond to 45.3% of the total and involve moving state and behavior among classes, as well as, superclass extraction in class hierarchies. We discuss the most interesting of these cases that are also characterized by design tradeoffs in Section 4.1.4.1.

**Tab. 4.5.** Metric fluctuations in interesting cases

| Id | Scenario | Commit | WMC | LCOM5 | CBO | DIT |
|---|---|---|---|---|---|---|
| R1 | 3 | 4c2a050 | 10 | 3 | 25 | 18 |
| R2 | 3 | 74a5c5d | 4 | 2 | 2 | 0 |
| R3 | 4 | 1707a94 | -9 | -1 | 5 | 2 |
| R4 | 4 | 202f00e | 1 | 0 | -1 | 0 |
| R5 | 4 | 528da74 | -2 | -2 | 1 | -1 |
| R6 | 4 | efd8856 | 12 | -3 | 0 | 0 |

#### 4.1.4.1. *Interesting cases*

Our manual evaluation of revisions revealed several design decisions related to the refactorings that we detected. In this section, we select and explain interesting design decisions identified in RCs from Scenarios 3-4. Moreover, we discuss the effect on internal metrics of the refactorings applied in each revision. We summarize these revisions in Tab. 4.5. Each revision is given a number, which we use in the rest of the text for identification. Further, for each one, in Columns 2–7, we list under what scenario it was classified, its abbreviated Git Commit ID and the aggregate metric differences. For each revision, we display its key take-away in a boxed sentence.

The first two revisions were classified in Scenario 3 and include some interesting design decisions. The remaining four revisions were classified in Scenario 4. One of these revisions, R3, involves one of the most complex refactorings in the revision history of JFreeChart.

**Revision R1**. In this revision, an Extract Superclass refactoring unifies under a common parent, the `TextAnnotation` and `AbstractXYAnnotation` class hierarchies, as well as the individual class `CategoryLineAnnotation`. This way, a larger class hierarchy is formed having the extracted superclass `AbstractAnnotation` as root. The refactoring was motivated by the need to add an event notification mechanism to plot annotation classes[3]. The developers decided to add this feature to all plot annotation classes through its implementation in a common superclass (`AbstractAnnotation`). The implementation comprises appropriate state variables and methods for adding/removing listeners and firing change events. The

[3]`https://sourceforge.net/p/jfreechart/patches/253/`

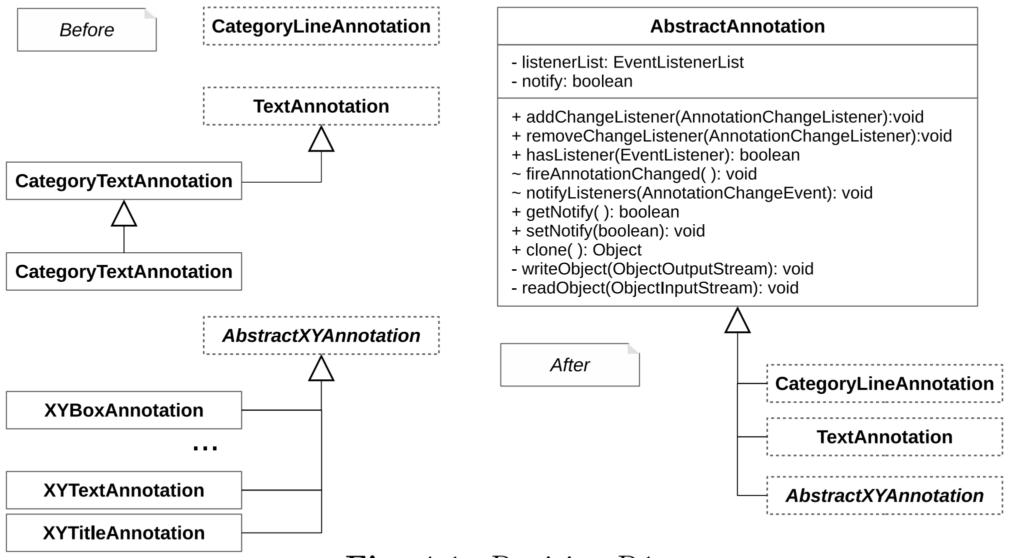

Before
CategoryLineAnnotation
TextAnnotation
CategoryTextAnnotation
CategoryTextAnnotation
AbstractXYAnnotation
XYBoxAnnotation
...
XYTextAnnotation
XYTitleAnnotation
AbstractAnnotation
- listenerList: EventListenerList
- notify: boolean
+ addChangeListener(AnnotationChangeListener):void
+ removeChangeListener(AnnotationChangeListener):void
+ hasListener(EventListener): boolean
~ fireAnnotationChanged( ): void
~ notifyListeners(AnnotationChangeEvent): void
+ getNotify( ): boolean
+ setNotify(boolean): void
+ clone( ): Object
- writeObject(ObjectOutputStream): void
- readObject(ObjectInputStream): void
After
CategoryLineAnnotation
TextAnnotation
AbstractXYAnnotation

**Fig. 4.1.** Revision R1.

new feature increased the DIT value of all `AbstractAnnotation` subclasses, as well as their coupling (CBO) due to invocations of inherited methods. The negative impact on WMC and LCOM5 metrics is due to extra functionality added to client classes of the new feature (e.g. `Plot`, `CategoryPlot`).

Revision R1 shows an occurrence of a design decision that spans over multiple classes where there is no tradeoff with respect to metrics.

**Revision R2**. This revision involves two Pull Up Method refactorings from `AbstractCategoryItemRenderer` to the parent class `AbstractRenderer`. The refactorings enable reuse of functionality related to adding rendering hints to a graphics object. The functionality was introduced in a previous revision to `AbstractCategoryItemRenderer` and is reused in order to provide hinting support to all renderers. In revision R2 the methods are invoked from `AbstractXYItemRenderer` and its subclass `XYBarRenderer`. The refactorings added extra methods to `AbstractRenderer` and, thus, increased the values of WMC, LCOM5 and CBO metrics. Although metric values were improved (negative change) for `AbstractCategoryItemRenderer`, the aggregate change values for the revision are still positive due to method declarations and invocations in `AbstractXYItemRenderer` and `XYBarRenderer`.

Revision R2, while very similar to R1, shows that the direction of changes happening at the class granularity can be masked by the revision granularity in the same design decision.

**Revision R3**. This revision includes 20 refactoring operations comprising 1 Extract Superclass, 1 Extract Method, 1 Rename Method, 10 Pull Up Attribute and 8 Pull Up Method. The refactoring inserts an intermediate subclass (`DefaultValueAxisEditor`) between `DefaultAxisEditor`, the hierarchy root, and `DefaultNumberAxisEditor`, its direct child. The new parent of `DefaultNumberAxisEditor` absorbs a large part of its state and behavior. The refactoring was motivated by the need to introduce a properties editing panel for the logarithmic scale numeric axis. The new panel (`DefaultLogAxisEditor`) has overlapping functionality with `DefaultNumberAxisEditor`. This functionality is reused through inheritance and `DefaultLogAxisEditor` is implemented as a subclass of `DefaultValueAxisEditor`. Moreover, the developers decided to reuse `DefaultNumberAxisEditor` functionality through a new parent class, in order to maintain the abstraction level of the hierarchy root. However, the CBO and WMC of `DefaultAxisEditor` have increased, since it, also, serves as a factory for creating instances of its subclasses. The positive impact on metrics in revision R3 (reduction of WMC, LCOM5) is dominated by the simplification of the `DefaultNumberAxisEditor` implementation due to pull up refactorings.

Revision R3 shows a design decision spanning over multiple classes where there is a trade-off in metrics. Moreover, it shows that examining metrics at revision level can mask important details happening in smaller levels. Additionally, this is an example of tangled commit where an implementation is also added for PolarPlot editor.

**Revision R4**. The focus of code changes in this revision is the simplification of the API that `Plot` class provides to its subclasses. The applied refactorings extract the notify listeners functionality to a new method, `fireChangeEvent`() with protected visibility. Although the implementation of the extracted method is rather simple, it replaces the notification logic in fifteen locations in the `Plot` class and in several locations in its subclasses `CategoryPlot`, `FastScatterPlot` and `XYPlot`. Moreover, it decouples `Plot` subclasses from the implementation of the change event. The refactoring increases the WMC of `Plot` due to the new method declaration and decreases the CBO of its subclasses due to the removal of references to the change event implementation (`PlotChangeEvent`). We note that due to unused imports of the `PlotChangeEvent` class in `Plot` subclasses, the SourceMeter tool does not recognize the reduction of CBO in all cases.

Revision R4 shows a design decision affecting multiple classes where the trade-off is between two metrics only. Additionally, this is a revision where there is no granularity conflict between revision and classes. This represents a best case: the revision contains only the refactoring implementation which corresponds to a single design decision that is represented by a metric trade-off.

**Revision R5**. In this revision, the identified refactorings involve moving an attribute and two methods, relevant to rendering a zoom rectangle, from `ChartViewerSkin` to `ChartViewer` class. The `ChartViewerSkin` is removed from project and `ChartViewer` is turned from a UI control to a container for the layout of chart canvas and zoom rectangle components. The simplification of `ChartViewer` is responsible for the improvement of WMC, LCOM5 and CBO in revision R5. However, the CBO improvement has been counterbalanced due to another refactoring, not detected by RMiner, that implements a second design decision within the same revision. The refactoring involves the move and inline of two `ChartCanvas` methods in the `DispatchHandlerFX` class. The methods are related to dispatching of mouse events and their relocation introduces a *Feature Envy* code smell in `DispatchHandlerFX` and respective increase in the CBO metric. Nevertheless, this solution is preferred since it enforces a basic decision in the design of ChartCanvas: its behavior related to user interaction should be dynamically extensible through registration of `AbstractMouseHandlerFX` instances.

Revision R5 shows two design decisions affecting multiple classes resulting in a classification into Scenario 4. If only one design decision where to have been implemented, it would have been categorized as Scenario 3.

**Revision R6**. Finally, this revision includes a Move Method refactoring from `SWTGraphics2D` to `SWTUtils`. The refactoring enforces the decision that reusable functionality related to conversions between AWT and SWT frameworks should be located in `SWTUtils` class. The move method lowers the complexity and improves the cohesion of `SWTGraphics2D`, although its WMC value is not changed due to extra functionality added in the same revision. On the other hand, the cohesion of `SWTUtils` is slightly changed contributing, thus, to the tradeoff between WMC and LCOM5 at revision level.

Revision R6 shows a design decision paired with a feature implementation creating an opposite change for one metric at the class granularity.

In-depth inspection of revisions R1–R6 lead us to noteworthy observations on the presence of design decisions and the hints that refactorings and metric fluctuations provide for their identification. First of all, combined fluctuations of DIT and CBO within revisions, as is the case in {R1, R3} provide evidence of structural changes potentially related to design decisions. The type and target of refactoring operations can contribute to tracing the classes affected by these decisions. On the other hand, fluctuations of WMC and LCOM5 metrics, usually indicating changes to class responsibilities, provide a strong indication of design decisions when they cause tradeoffs with other metrics (e.g, R3–R5). However, the impact of refactorings to fluctuations of WMC and LCOM5 is often obscured by code additions that implement new features. The problem is exaggerated in RCs with tangled changes, as is the case in {R3, R5}.

We also observe that interesting decisions are identified in RCs that implement new features. Moreover, given that refactorings R4–R6 enforce past decisions, the identification of recurrent patterns of past revisions provide stronger hints on the importance of decisions.

In conclusion, the qualitative analysis gives us confidence that the phenomenon is real and worth further inspection.

## 4.2. Exploring Tradeoffs

A key component of our approach is the notion of *quality tradeoffs*, i.e., changes where developers consciously prioritize some internal code quality characteristics at the expense of others. In practical terms, these revisions would include both metrics that have improved and metrics that have deteriorated, due to the changes. Our argument is that they are an indicator of design activity. Our underlying assumption is that when contributors make tradeoffs between quality characteristics, they are deliberately or inadvertently expressing specific *design choices* in code.

In this section, we aim to answer RQ1, RQ2 and RQ3 with an empirical method in order to characterize metric fluctuations. We begin by specifying our methodology, then we describe our data collection process and the steps taken to analyze the data. Finally, we discuss our results and introduce the concept of *quality tradeoffs*.

### 4.2.1. Refined classification

Using the approach described in Chapter 3 and the results from Section 4.1, we introduce a classification taxonomy with two dimensions to characterize the changes in a commit at a macroscopic level. The taxonomy was further consolidated as the full dataset was collected.

A classification is defined as a tuple from the space $\Omega := (\mathcal{C}, \mathcal{D})$ where:

- $\mathcal{C} := \{\text{ZERO}, \text{ONE}, \text{MANY}\}$ representing the cardinality of changed metrics.
- $\mathcal{D} := \{\text{NEUTRAL}, \text{IMPROVE}, \text{DECLINE}, \text{MIXED}\}$ representing the direction of change for the changed metrics.

For the cardinality of change, we define ZERO when there is strictly no metric that changed for a revision; ONE when exactly one metric changed for a revision; and MANY when multiple metrics changed for a revision. For the direction of change, we define IMPROVE and DECLINE when all metrics go in a positive or negative direction with respect to quality respectively; NEUTRAL when the metrics of the artifacts change but balance out to zero during aggregation ; and MIXED for cases where some metrics IMPROVE and others DECLINE. The combination (ZERO, NEUTRAL) is a special case that signifies nothing changed. For example, the classification (ONE, IMPROVE) means that there was exactly one positive fluctuation of metric. Note that improvement or decline is defined with respect to quality, not value. Hence an increase in the value for, e.g., CBO, is measured as DECLINE, since increased coupling signifies quality deterioration.

We characterized every revision from Section 3.3 with the classification scheme proposed above. We present the results for each project in Fig. 4.2 and overall (average and cumulative) in Fig. 4.3 . The results are shown as heatmap tables, where the rows correspond to $\mathcal{C}$ and the columns to $\mathcal{D}$. Darker shades indicate higher numbers, whereas thatched lines indicate invalid combinations. The cells (ZERO, [IMPROVE, DECLINE, MIXED ]) and (ONE, MIXED) are hatched because these categories are impossible to fulfill: it is not possible to have zero metrics that improve, decline, or worse, improve and decline at the same time or have one metric that is simultaneously positive and negative.

Comparing the per-project and overall heatmaps, we observe that our distribution of categories is relatively uniform with no major variations between projects for each category. Indeed, the heatmaps in Fig. 4.3a and Fig. 4.3b are the same. Individual project heatmaps

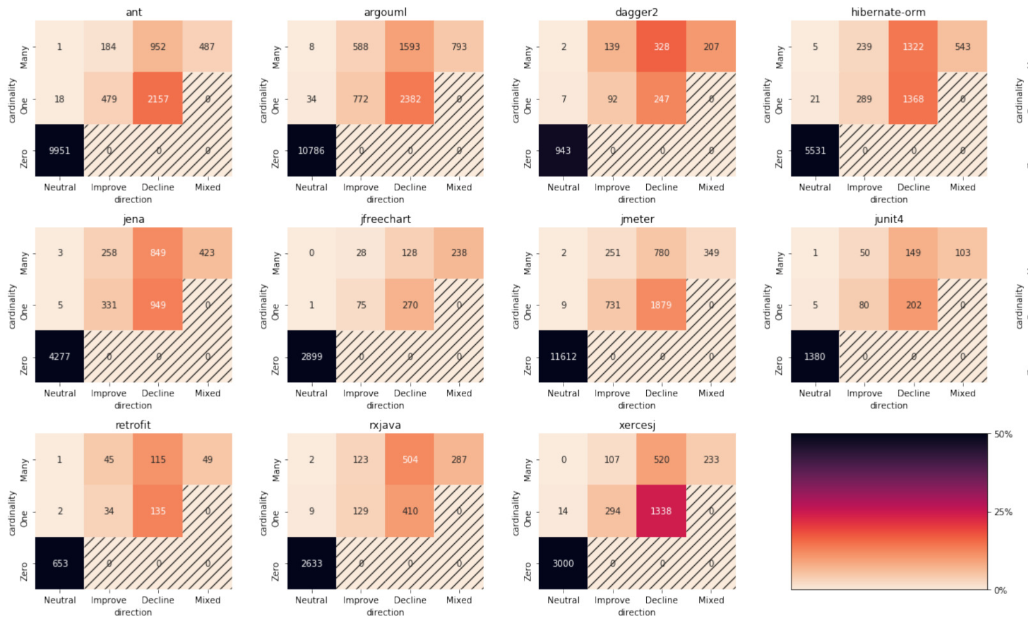
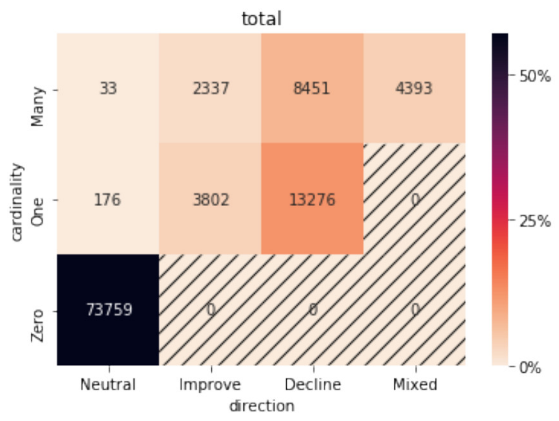
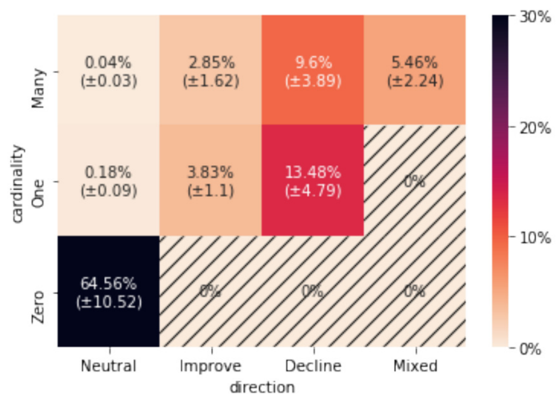


**Fig. 4.2.** Metric fluctuations for each project



**(a)** Cumulative sum



**(b)** Average

**Fig. 4.3.** Distribution of metric fluctuations in production code artifacts, averaged over all projects. The number in parentheses is the standard deviation.

in Fig. 4.2 follow the same pattern. Based on these observations we formulate for the studied metrics, there exist observable metric fluctuations between versions of a project.

We observe some recurring tendencies: First, we note that (ZERO, NEUTRAL) always holds the majority of revisions and represents 64.56% of occurrences on average. This means that, most of the time the four metrics we analyzed are not touched at all by the changes made in existing classes.

Second, we observe that the case where metric fluctuations cancel each other out in a revision represents only a small fraction of the overall observations. In this case, artifact metrics fluctuate but the aggregation process masks the effect (i.e., the net metric fluctuation for a revision sums up to 0). This case is represented by the revisions that were classified in the cells $\{\text{ONE}, \text{MANY}\} \times \{\text{NEUTRAL}\}$.

Third, we note that the group of four cells $\{\text{ONE}, \text{MANY}\} \times \{\text{IMPROVE}, \text{DECLINE}\}$ is well populated in all projects and that observations in these cells represent 29.76% of the occurrences. In these cells, the overall quality of the code (as viewed through metrics) is either monotonically improved or worsened in one or more aspects. While the distribution of occurrences between each of the four cells is variable, we can see that there are always more DECLINE occurrences than IMPROVE occurrences. Keeping in mind that we track the fluctuations of only a subset of metrics and that we ignore the changes induced by new or deleted classes, this observation suggests that, over the development of these applications, classes tend to decline in quality more than they improve overall. This can be explained as a reflection of the fact that as a program grows with new capabilities, so does its intrinsic complexity and size, which seems to be evidence for the design erosion phenomenon [**82**].

Fourth, we observe a significant minority of occurrences clustered in the cell (MANY, MIXED). This cell represents cases when at least two quality metrics changed in opposite directions. It represents, on average, 5.46% of the occurrences, which is about a fifth of the cases where metrics change monotonically. These cases are notable because they capture revisions where code changes improve the codebase in some quality aspects at the cost of others. We consider this fluctuation pattern as a reflection of *tradeoffs* between metrics.

In summary, we found that (a) in the majority of revisions, there are no metric fluctuations; (b) metric fluctuations rarely "cancel out"; (c) metrics change monotonically in roughly a third of all revisions; (d) quality deteriorates more often than it improves; (e) metric tradeoffs exist in a minority of revisions.

Based on observation (a), we can confirm that, at least from a metrics perspective, in the largest part of a project's history there is very little happening that is significant to the measurable design quality of the project. By extension, for most of a project's history, measurable quality characteristics (i.e., metrics) cannot help developers understand what quality attributes are of greater priority to the project. However, observation (e) leads us to conclude that there consistently exists a small number of revisions in which there are observable and *measurable* quality tradeoffs. These can be indicative of design activity, and can potentially reveal a great deal of information about the design decisions and principles followed in a project. Narrowing the scope to these signs of activity for future qualitative analysis of the system's design quality has the potential to provide insights for improving project awareness, onboarding and overall developer productivity. It is important to mention that, even though we focused on illustrating fluctuations along two dimensions (direction, cardinality), our approach can be easily adapted to take into account further characterization dimensions, such as the magnitude of the metrics fluctuations.

### 4.2.2. Fluctuations in context

In this section, we address RQ3 by analyzing our dataset under the lens of three separate development contexts: (a) production versus test code, (b) refactoring, and (c) the different phases of the release cycle. We compare how the categorization of revisions according to their metric fluctuations changes for each of the contexts. Further, as our exploratory analysis pointed to the existence of revisions with measurable quality tradeoffs, we specifically focus our investigation on understanding how such tradeoffs fit with existing theories about each of the studied contexts.

#### 4.2.2.1. *Context 1 – Production vs. Test Code*

We want to compare the distributions in metric fluctuation categories for revisions that affect production code versus those that affect test code artifacts. Tests can be considered a form of specification of the requirements of a software project. We thus expect their quality to be more stable than that of production code as shown in Fig. 4.3. Further, if revisions whose metric fluctuation matches the tradeoffs category are indeed signs of design activities, we expect that test code should have relatively fewer revisions in this category, as design is an effort that mostly concerns production code.

To make the comparison, we modify the *Calculating changes (E)* analysis step described in Chapter 3 in order to include test artifacts. For each commit, we creates to sets of changes, one for the the production code and want for test code We then continue the rest of the analysis procedure for each set of metric fluctuations, resulting in two sets of classification for all revisions: $\omega_p$ and $\omega_t$, for production and test code respectively.

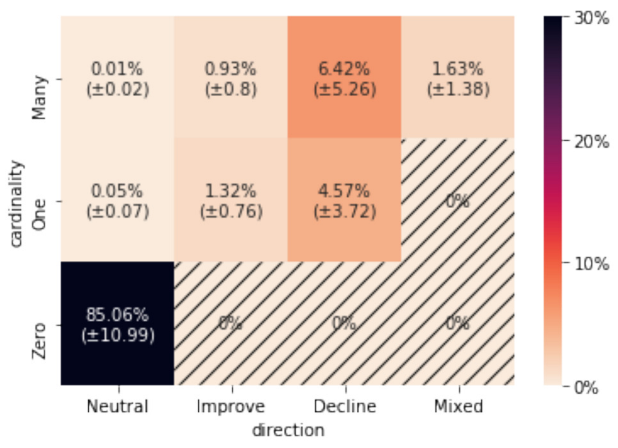


**Fig. 4.4.** Distribution of metric fluctuations in test code artifacts, averaged over all projects. The number in parentheses is the standard deviation.

The heatmaps representing the distribution of metrics fluctuations for production and test code are shown in Fig. 4.3b and 4.4, respectively. We note that the test distribution also follows the trend we saw in Sec. 4.2.1 where the majority of revisions are categorized as (ZERO, NEUTRAL), i.e., the metrics do not change. However, we also note that test code has a higher percentage of revisions in this category (85.06% vs 64.56%). This is consistent with our hypothesis that the quality of test code is more stable than that of production code. We also note that production code has a significantly larger percentage of revisions in the (MANY, MIXED) category, which is consistent with the interpretation that tradeoffs should concern production rather than test code. As previous results suggested, we observe that NEUTRAL changes for ONE or MANY changes represent a negligible number of revisions. Finally, in the DECLINE column of both heatmaps, design erosion [**82**] is clearly observable.

In conclusion, we see that the change of code context between production and test has a noticeable effect on the classification of the metric fluctuations.

### 4.2.2.2. *Context 2 – Refactoring*

We want to compare the distribution in metric fluctuation categories for RCs, compared to the general case presented in Section 4.2.1.

Refactoring has been extensively studied for its impact on software quality [**1, 74**]. Researchers have focused on the identification of refactoring opportunities and on analyzing

**Tab. 4.6.** Ratio of revisions containing a refactoring to total number of revisions

| Project | Refactorings | Ratio |
|---|---|---|
| Ant | 1 702 | 11.96% |
| ArgoUML | 2 001 | 11.8% |
| Dagger2 | 568 | 28.91% |
| Hibernate ORM | 1 517 | 16.28% |
| jEdit | 1 394 | 6.1% |
| Jena | 1 125 | 15.86% |
| JFreeChart | 159 | 4.37% |
| JMeter | 1 403 | 8.99% |
| JUnit4 | 332 | 16.85% |
| OkHttp | 467 | 23.99% |
| Retrofit | 191 | 18.47% |
| RxJava | 608 | 14.84% |
| Apache Xerces-J | 648 | 11.77% |

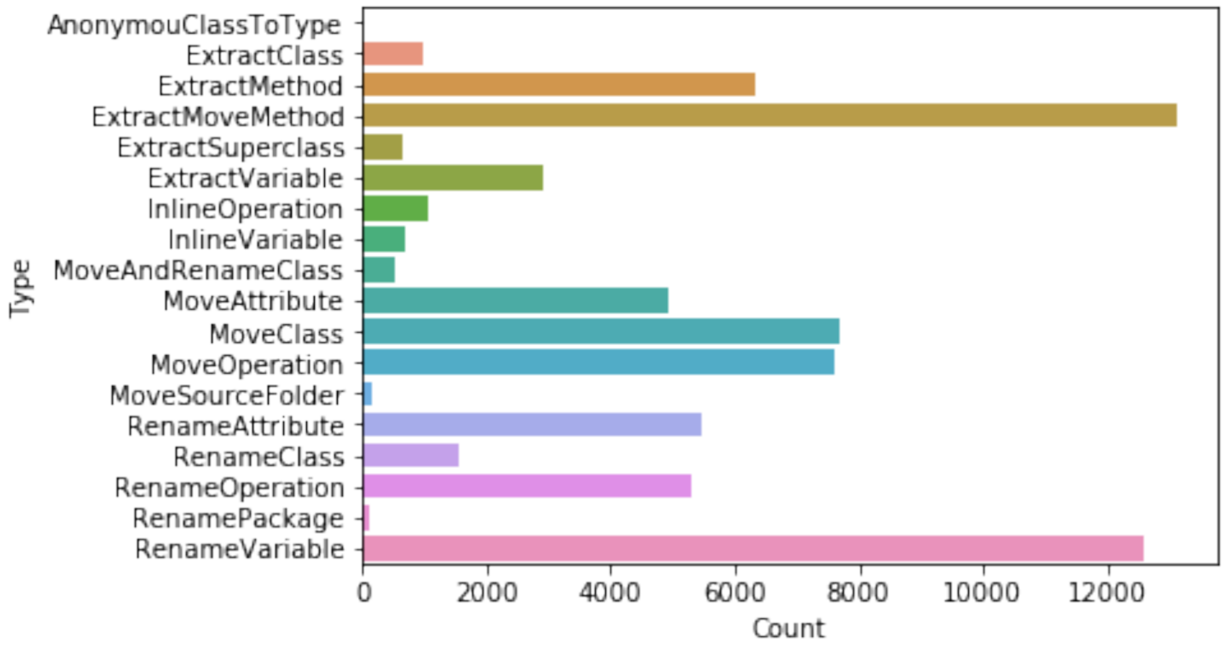


**Fig. 4.5.** Types of refactoring

their impact on design, using code smells [**29, 79**] or code metrics [**53**]. The resulting consensus is that refactoring impacts the design quality of a system in a significant way. This relationship is not an incidental: developers purposefully use refactoring to express specific design intentions [**70**]. We thus expect that there should be relatively fewer RCs that have no impact in metrics.

Refactoring activity can carry various kinds of design intent, including but not limited to: removal of code smells, resolution of technical debt, application of design principles, and introduction of design patterns. Additionally, we know that changes in design, positive or negative, are reflected in software metrics [**44**] and that refactoring doesn't always improve monotonically the quality of an application. In other words, developers consciously make design quality tradeoffs while refactoring. It is therefore a very interesting context in which to study metric fluctuations. We thus also expect to find an increased presence of tradeoffs among RCs.

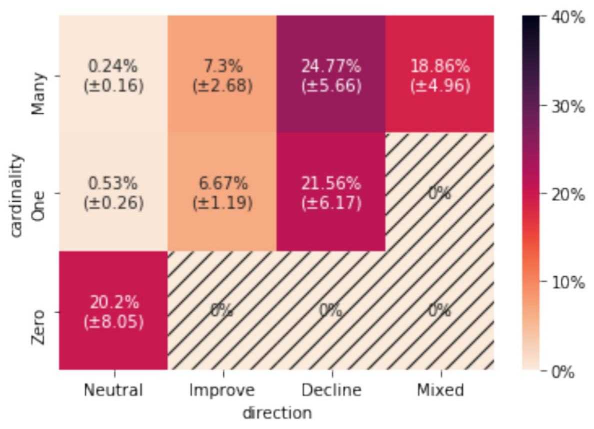
We filtered the 107 449 commits of all the projects, down to 12 115 RC, i.e., revisions with at least one refactoring. The median percentage of RCs in the entire revision history is 14.84%. The project with the lowest percentage is JFreeChart with 4.37% and highest is Dagger2 with 28.91%. This gap can have many causes including the accuracy of RMiner, the complexity of refactorings used by the developers, and their development habits and guidelines. The ratio for each project is shown in Table 4.6.

The heatmap showing the distribution of metric fluctuations for RCs, averaged for all projects is shown on Fig. 4.6(a). We show the percentage of RCs compared to the total number of revisions in each category in Fig. 4.6(b). It is immediately evident that the distribution is very different from the one in Fig. 4.3(b).

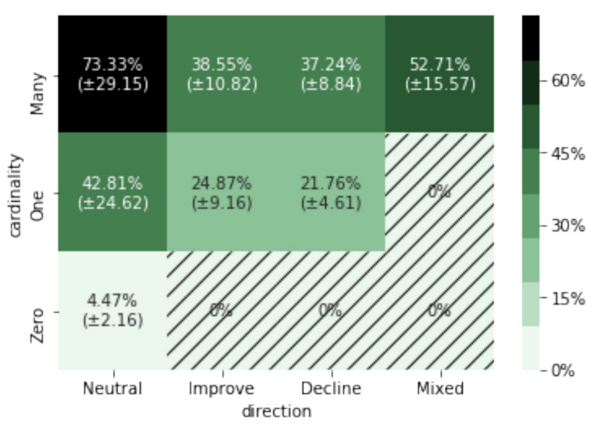
First we note that in the context of refactoring the category (ZERO, NEUTRAL) is notably less populated, as on average only 20.2% of RCs fall in this category. We can also see in Fig. 4.6(b) that RCs are more than 50% of the total number of (MANY, MIXED) revisions. These observations are consistent with our hypothesis that RCs are more likely to be affecting design quality.

Second, we note that there are about four times as many RCs that are categorized as (MANY, MIXED) and are thus potentially design activities because of tradeoffs in metrics. This is also consistent with our hypothesis that refactoring is an activity during which developers are bound to be making design quality tradeoffs. Similarly to the previous context, we also observe design erosion.

We thus conclude that the refactoring context has a big effect on metric fluctuations.



(a) Average percentage of metric fluctuations for each category.



(b) Average percentage of RCs with respect to all revisions for each category.

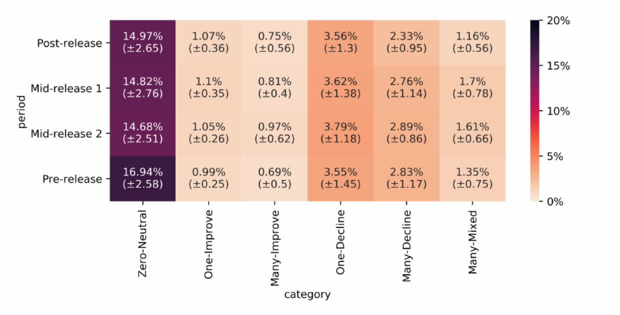
**Fig. 4.6.** Metric fluctuation distributions for RCs averaged for all projects. The number in parentheses is the standard deviation of each average.

#### 4.2.2.3. *Context 3 – Releases*

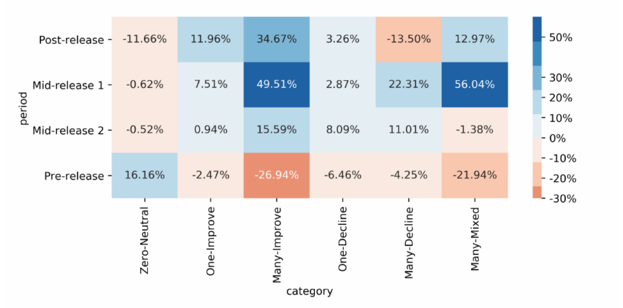
We want to understand how the distribution in metric fluctuation categories for revisions changes relative to the development cycle. Projects are often put in a "freeze" state before important releases [**59**]. Moreover, Potdar et al. have found that little self-admitted technical debt is accrued close to release dates [**65**]. We expect this to be reflected in quality metric fluctuations closer to release dates.

In order to study the distribution of metric fluctuations within releases, we partition of the set of revisions $v$ of each project into a set of *release slices* $v_i$. Each *release slice* $v_i$ represents a part of the revision history that contributed to project evolution between successive releases $R_{i-1}$ and $R_i$.

To compute the release slices for each project, we manually inspected all Git tags that annotate specific revisions in its evolution. Git tags are used to mark important points in a revision history and tags for release points usually follow a semantic versioning syntax scheme (e.g., v1.0, 2.1, r2.3). After identifying release tags, we manually sorted them in ascending order of Major, Minor and Patch numbering. Next, we processed pairs of successive releases $(R_{i-1}, R_i)$ to identify the slice of commit history that contributed to transition of the project from release $R_{i-1}$ to $R_i$. The *release slice* thus consists of the set of reachable commits from tag $R_i$ after excluding the commits that are reachable from tag $R_{i-1}$. Notice that the set of



**(a)** Average percentage of project revisions over release periods and metric fluctuation categories. The number in parentheses is the standard deviation.



**(b)** Percent change of average ratio over successive release periods. Post-release is calculated relative to Pre-release.

**Fig. 4.7.** Distribution of metric fluctuations relative to stages in release cycle, averaged over all projects.

reachable commits of a revision represents all its ancestors in the commit history. The last commit in each release slice $\vec{\omega}_i$ is the one that is tagged with the release tag $R_i$. The release slice for $R_i$ can be generated by the following command: `git log` $R_{i-1}$`..`$R_i$ `-pretty="%H" -no-merges`

We manually recovered the release history of 10 projects in our dataset. We excluded ArgoUML, jEdit and JFreeChart due to low availability of release information in Git tags. We set the granularity of our analysis at the level of minor releases. In the projects we analyzed, patch-level releases were either too granular or too sparsely used, while major releases were too infrequent.

Within each release slice, revisions are further divided into groups, on the basis of their proximity to the release date. Given a release with tag $R_i$, its release date corresponds to the author date of the commit that is referenced by $R_i$. Thus, the proximity to release date for a commit $\omega_{i,j}$, belonging to release slice $\vec{\omega}_i$, is equal to the difference in days of its author date from release date. We divide each release slice $\vec{\omega}_i$ to four periods, demarcated by the quartiles of the proximity to release date of commits included in $\vec{\omega}_i$. For a release slice $\vec{\omega}_i$, we define the following periods:

- *Post-release*: commits with proximity to release that is larger than the third quartile Q3 (75th percentile) of proximity values within $\vec{\omega}_i$. These commits correspond to the

**Tab. 4.7.** Analyzed minor releases for each project

| Project | Release Range | | | Commits |
|---|---|---|---|---|
| | Start | End | Count | |
| Ant | ANT_12 | ANT_1.10.0_RC1 | 9 | 13225 |
| Dagger2 | dagger-parent-1.0.0 | dagger-2.20 | 24 | 1839 |
| Hibernate ORM | 4.0.0.Final | 5.4.0 | 9 | 7654 |
| Jena | jena-2.10.0 | jena-3.10.0 | 15 | 6960 |
| JMeter | v2_7 | v5_0 | 13 | 7572 |
| JUnit4 | r4.6 | r4.13-beta-2 | 8 | 1815 |
| OkHttp | parent-1.1.0 | parent-3.12.0 | 26 | 1811 |
| Retrofit | parent-1.0.0 | parent-2.5.0 | 16 | 944 |
| RxJava | 0.5.0 | v2.2.0 | 23 | 4403 |
| Apache Xerces-J | Xerces-J_1_1_0 | Xerces-J_2_12_0 | 17 | 5470 |

first revisions of a release slice and can be regarded as the first commits that follow release $R_{i-1}$ (hence the term *Post-release*).

- *Mid-release 1*: commits with proximity to release that is larger than the second quartile Q2 (median) of proximity values and less or equal to Q3. *Mid-release 1* follows the post-release period and can be regarded as a period when concerns handled by the release start to become more mature towards the pre-release period.
- *Mid-release 2*: commits with proximity to release that is larger than the first quartile Q1 (25th percentile) of proximity values and less or equal to Q2.
- *Pre-release*: commits with proximity to release that is less or equal to Q1. These commits represent the last commits of the release slice that lead to release $R_i$.

We implemented the retrieval of release slices and release periods for commits as a feature of MetricHistory. We summarize the details of the data collected in Tab. 4.7. Columns 2–3 show the first and last minor releases, while columns 4–5 show the total number of releases analyzed for each project and the overall number of commits. We exclude the first minor release of each project (as they have no "post release" phase) and releases with made beyond 2018-12-31.

The heatmaps representing the average ratio of project revisions over metric fluctuation categories and release periods is shown in Fig. 4.7(a). Each cell with row label $R_i$ and column label $C_j$ provides the average and standard deviation over all projects, for the ratio of project revisions that are committed in release period $R_i$ and classified as category $C_j$. For brevity we omit categories with very few revisions ((ONE, NEUTRAL), (MANY, NEUTRAL)).

We observe that in Fig. 4.7(a) the fluctuations in metrics do not change dramatically across the different release periods. The overall fluctuation pattern is that the majority of revisions are in the (ZERO, NEUTRAL) category, with a noticeable design erosion effect indicated by the DECLINE categories, and a small minority of (MANY, MIXED) revisions. We do however note a small observed increase in the (ZERO, NEUTRAL) category closer to the release date, which is consistent with the conclusions of published research mentioned above.

To further understand these observations, we also calculated the percent change of average ratio for each category of metric fluctuations over successive phases. The resulting heatmap is shown in Fig. 4.7(b). This representation shows more clearly the trends, normalizing the domination of the (ZERO, NEUTRAL) category. We make a few observations. First, we note that the increase in the (ZERO, NEUTRAL) category in the pre-release period is coupled with a decrease in all other categories. This is again consistent with published research. Second, we note that there is a marked increase in all other categories during the first mid-release period. One explanation could be it is during this period that the project is most in flux, with feature additions and maintenance tasks. It is notable that the (MANY, MIXED) category is most increased during this period. This might indicate that future research should focus in this period to identify quality tradeoffs.

We conclude that the effect of the release context to metric fluctuations is not big. However we can be observe some interesting differential effects.

### 4.2.3. Discussion

We have studied metric fluctuations in three contexts. First we compared the fluctuations in production and test code and found that there is a noticeable difference between the two, with test code being more stable and containing fewer tradeoffs. Second, we investigated refactoring and found that it has a clearly identifiable effect on metric fluctuations. Third, we studied how metric fluctuations change over time during releases. We found a very small effect closer to release dates, as well as some interesting differential effects. These observations lead us to formulate an answer to RQ3 that fluctuations in metrics can depend on the development context. Future research should investigate additional contexts and fluctuation categories, as well as differential effects.

Our study found that quality tradeoffs coincide with particular and significant development activities and milestones. We also found that, although they carry a lot of information about an artifact, quality tradeoffs are the minority of metric fluctuation patterns in commits, which actually confirms our premise that developers need only focus on a subset of the artifact's history to understand. This study is the first step towards improving (a) the productivity of developers, by reducing the size of the material that needs to be studied to contribute to a component, and (b) the understanding of developers about a project.

## 4.3. Towards Design Intent

In this section, we conduct a preliminary study for RQ4. We are interested to see if there is a correlation between internal quality metric fluctuations in RCs and design intent.

To study this question, we manually analyze 106 commits. These commits are sampled from the RCs of eleven projects (Hibernate ORM and Apache Xerces-J data were not yet mined when we conducted this study)[4]. For each project, we selected randomly up to 10 commits classified as (MANY, MIXED) (corresponding to Scenario 4). The goal is to identify the design intent behind the applied refactorings; we determine whether in each RC in the sample the developer applied the identified refactorings as part of introducing new design decisions or enforcing design decisions that were established in previous commits.

To better understand the context in which these activities happen, we also wanted to find out the type of implementation task the developer was engaged in while refactoring, i.e., whether any design decisions were enforced as part of refactoring low quality code, implementing new features, or fixing bugs.

### 4.3.1. Setup

We based our analysis on code and comment inspection, commit messages, and the changelog of refactored classes. Using this information, we recorded for every RC in the sample (b, c) whether the proposition "the revision carries design intent" holds, and (d) the task type (refactoring, feature implementation, or bug fix). To calibrate the manual analysis,

[4]JFreeChart only had 6 commits conforming to this classification as described in the first preliminary study in Section 4.1.

V. Zafeiris and T. Schweizer independently analyzed all RCs found by RMiner in JFreeChart. The inter-rater agreement between their assessments was moderate, indicated by a value of 0.490 for Cohen's Kappa and the percentage of observed agreements was 73.53%. Then, we did a consolidation phase, where we defined a common annotation protocol (described below). The protocol was then used to analyze the rest of the RC.

Specifically, we defined and followed the following annotation protocol for each RC:

**Find relevant entities (a):** First, we identify all refactorings in the RC and the entities they affect. This information is generated by RMiner. Unless they are related to other structural changes, we ignore Rename refactorings because we are interested in the intent behind decisions affecting the structural design of the software.

**Decide if the refactoring itself implements a design decision (b):** Then, we decide if at least one of the refactorings in the RC is an expression of a design decision. For example, the developer may decide to delegate some functionality to a utility class using a Move Method refactoring. To determine whether a refactoring (or a set of refactorings) enforce a past design decision, we also trace back to previous revisions of the refactored code to understand the evolution of the design.

**Decide if the refactoring is part of a design decision (c):** In this step, we decide if at least one of the refactorings in the RC is used to implement a wider-range design decision that primarily affects another code entity (i.e., an entity not affected by a refactoring) in the same RC. For example, the developer may decide to introduce a new design pattern to regulate the communication between two classes and use Move Method as part of the implementation of this decision. To better understand the architectural role of each entity, we also first study the overall context of the project, its organization, structure, and business logic.

**Determine the task type (d):** Finally, we record the type of implementation task that the developer was involved in when they did the refactoring. We use the terms defined by Murphy-Hill et al. [**61**]: "root canal" describes revisions with a pure refactoring purpose, whereas refactorings that are part of the implementation of new features are termed "flossing". A refactoring can also be part of a bug fix. We detected the task type by inspecting code differences combined, and analyzing commit logs and

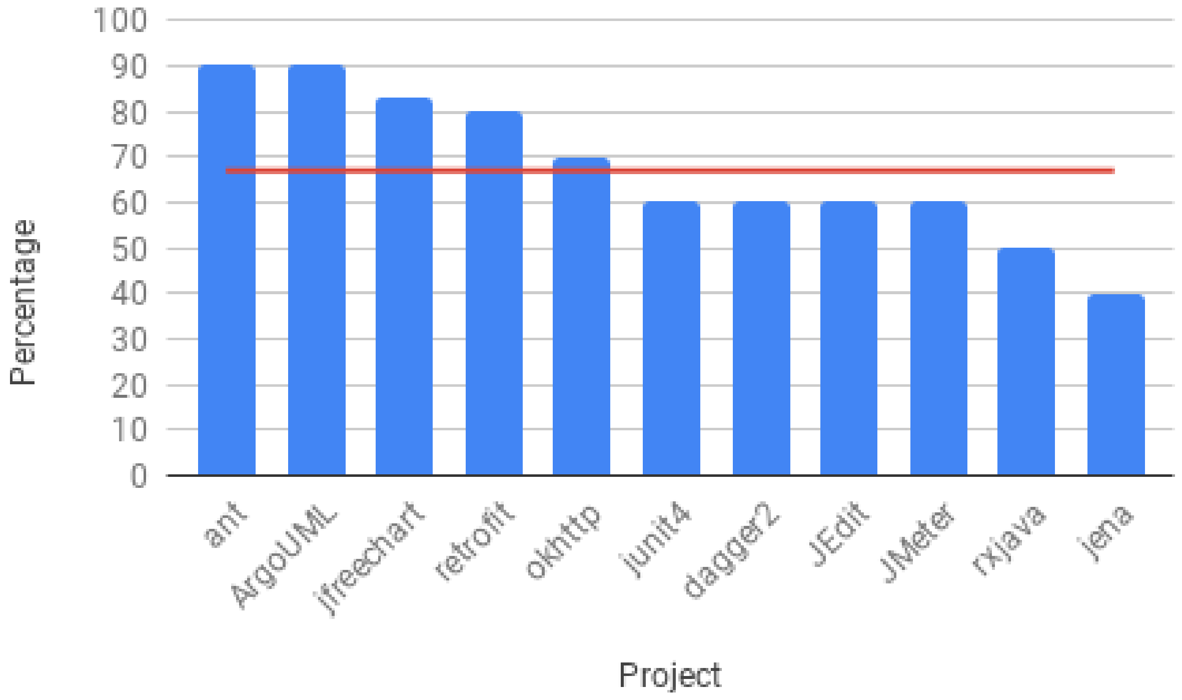


**Fig. 4.8.** Percentage of design decisions in each sample per project. The red line is the average number of detected design decisions across all samples.

**Tab. 4.8.** Distribution of the implementation tasks in the sampled set of RCs.

| Task type | Revisions | Design intent present |
|---|---|---|
| "Root canal" refactoring | 56 (52.83%) | 40 (37.74%) |
| "Flossing" (feat. implem.) | 33 (31.13%) | 24 (22.64%) |
| Bug fix | 17 (16.04%) | 7 (6.60%) |
| **Total** | 106 (100%) | 71 (66.98%) |

embedded change logs of refactored classes. In several cases, we took advantage of references to issue tracking identifiers in commit and change logs.

If in steps (b) or (c) we identify a design decision, we annotate the RC in the sample as containing a design decision.

### 4.3.2. Results

We found that 71 out of 106 total RCs in our sample carry design intent. That corresponds to an average 67% RCs out of all scenario 4 sampled RCs in the projects that we studied. With a confidence interval of 95%, the results should be taken with a margin of error of 8.6. We summarize our findings per project in Fig. 4.8. We observe a degree of variation in the percentages for each project, but for 10 out of 11 projects, the coincidence of metric tradeoffs and refactoring indicates the presence of design intent in more than half of the cases. For RxJava the percentage is 50% and for Jena 40%, making it the only exception. Jena is so low because the sampling gave RCs with a lot of Rename refactorings that are not directly tied to design decisions.

We summarize the types of implementation tasks that developers were involved in RCs in Tab. 4.8. “Root canal” refactoring corresponds to 52.83% of total revisions. “Flossing” refactoring corresponds to 31.13% of total revisions and involve moving state and behavior among classes, as well as superclass extraction in class hierarchies. If we only take into account the RCs containing decisions we can see that pure refactoring and feature addition contains the most design decisions while bug fixes has the lowest. Indeed, the Refactoring category has a percentage decrease of 28.57%, Feature Implementation has 27.27%, and Bug Fix has 58.82%.

We can see a concrete example in action in a commit submitted to JFreeChart on 2011-11-12 by the user `matinh`[5] and was classified as Scenario 4 by MetricHistory, computing the tuple $s$=($WMC$=-9, $LCOM5$=-1, $CBO$=5, $DIT$=2, $hit\_count$=4). This revision includes 20 refactoring operations: 1 instance of Extract Superclass, 1 of Extract Method, 1 of Rename Method, 10 of Pull Up Attribute, and 8 of Pull Up Method. The original version of the code contained the class `DefaultAxisEditor` and a subclass `DefaultNumberAxisEditor`. These classes implement panels in JFreeChart. In this revision, the developer wants to add a new panel class, `DefaultLogAxisEditor` that allows editing logarithmic axes. However, the functionality of this new class overlaps with the existing class `DefaultNumberAxisEditor`. To avoid duplication, the developer refactors the class inheritance hierarchy. He inserts a new intermediate subclass, `DefaultValueAxisEditor` between `DefaultAxisEditor` and `DefaultNumberAxisEditor`. Large parts of the state and behavior

[5]`https://github.com/jfree/jfreechart/commit/1707a94af46df84373e3118c62e18359d40b1f16`

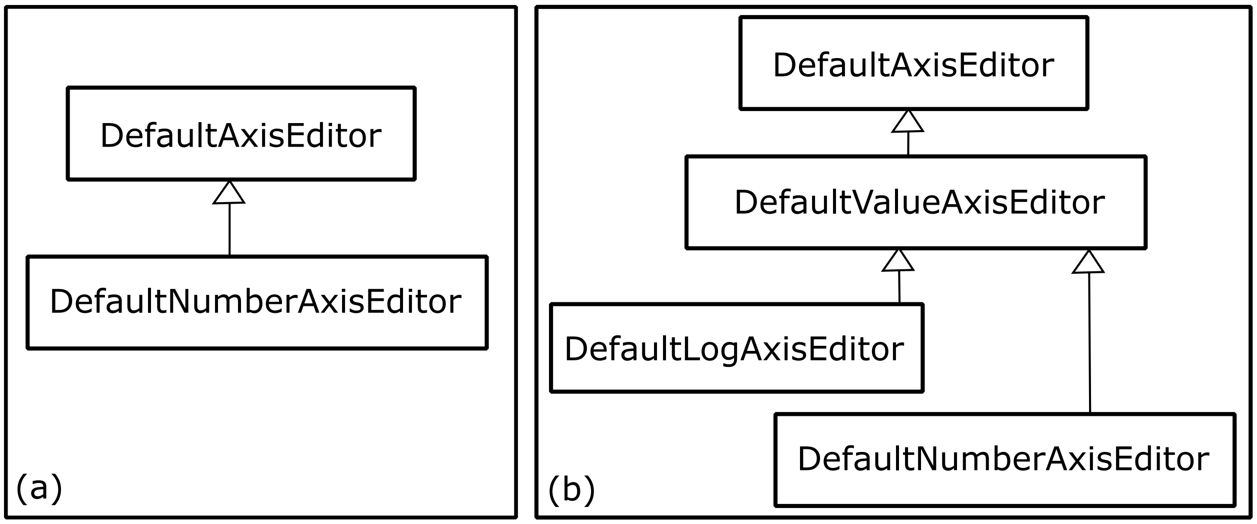


**Fig. 4.9.** (a) Slice of the JFreeChart design. (b) The same slice after the introduction of `DefaultLogAxisEditor` and the compound refactoring that created `DefaultValueAxisEditor`.

of `DefaultNumberAxisEditor` are then pulled up to the new class so it can be reused by `DefaultLogAxisEditor` through inheritance. We illustrate the two versions in the UML Class Diagram shown in Fig. 4.9.

At the revision level, the WMC (complexity) and LCOM5 (cohesion) metrics are improved (i.e., reduced) due to the simplification of the `DefaultNumberAxisEditor` class implementation caused by the pull up refactorings. On the other hand, the design becomes more complex, evidenced by the deterioration (i.e., increase) in the DIT (inheritance depth) and CBO (coupling) metrics. This is clearly the result of the developer's intent to incorporate the new class in the existing design.

In conclusion, this qualitative study confirms that a correlation between the four internal quality metrics and design intent exists to some extent in the context of RCs.

## 4.4. Threats to validity

**Construct validity** is threatened by multiple sources. First, by using only four metrics, we are bound to miss some aspects of changes in a RC. To mitigate this, we selected metrics that have been tied to well known internal quality attributes and that are general enough to capture a maximum of their respective code aspect.

Second, as mentioned in Chapter 3, we ignore the metric fluctuations generated from classes that are added or deleted. This means our pipeline does not capture metric fluctuations on the entire change set of a revision, in cases of class addition or removal. Interestingly, we observed no statistical differences in distributions for detecting tradeoffs when taking into account added and deleted classes.

Third, we concentrated on metric fluctuations at the class level. Doing that, we might have missed variations in metrics in smaller or larger granularities. To mitigate this, we count the number of metric changed when we aggregate the metrics at the class level to represent the revision. This way, we would see if a metric changed even if the changes of metric of two classes or more would cancel each other. For these reasons, we cannot draw general conclusions about what would happen at different granularity levels or with other metrics. Regardless, this does not impact the main contributions of this exploratory, i.e., the systematized methodology, classification scheme and research questions.

Fourth, we do not count Rename operations as design decisions in Section 4.1. One could argue that pure Rename refactorings (who constitutes the majority of scenario 1) also defines design decisions. After all, we use natural language to communicate intent. If the name of an entity change so should the intent behind the class. However, for this study we are only interested in structural or architectural design decisions.

Fifth, the initial inter-rater agreement presented in Section 4.1.3 is also a potential threat to validity since the reviewers were only in *moderate* agreement. This threat was mitigated by the conflict resolution procedure we conducted afterward to harmonize the datasets. Moreover, from this experience, we were able to improve our manual analysis procedures for subsequent usages such as Section 5.1.

Finally, the study is based on the analysis of arbitrarily-cohesive units of work determined by each developer. The content of a commit is determined by the developer. During our manual analysis, we saw different level of commit hygiene – how well a commit is made. Does it contain a good description? Are the changes separated? Some would neatly describe the changes and include strictly the relevant code changes. Others, would commit everything they changed since their last commit whether it was a few hours or a few weeks, resulting in very hard to read commits [**2**]. Incidentally these commits tended to also have the worst descriptions. In our context, this diversity is not very relevant since we're doing exploration.

However, while building a classifier it could become a problem and cause the model to miss classify commits. However, this is mitigated by the fact that big commits are likely to touch multiple concerns of the code and thus certainly affect design that will be reflected in the metrics. In a more fine grained approach where we want to locate the design changes, this would potentially be a problem. Approaches are being researched to untangled code changes in commits [**72**].

**Internal validity** is threatened by the off-the-shelf tools used in our data processing pipeline. Despite its high accuracy and recall, RMiner has a small chance to produce false positives and miss refactorings. To mitigate this factor, we combed through the refactorings of JFreeChart manually to remove false positives and assessed the gravity of the threat. We determined that RMiner was giving false positives in cases that were innocuous to the study. Allowing us to proceed for the other projects with confidence.

Additionally, we depend on SourceMeter for the calculation of metrics and are, therefore, tied to its quality. This threat will be mitigated in future reproductions of the study, since we architected our toolchain such that new versions of the tool or entirely different tools can be easily integrated.

**External validity** is threatened by the generalizability of the study. We analyzed 13 open source projects and focused on a subset of their version history. Thus, our study is biased by the development practices used in these projects. An argument towards the representativeness of the selected sample of RCs are the different levels of commit hygiene revealed by a preliminary manual analysis; some revisions were very well documented and worked toward a clear, unique, defined goal while some other had misleading, vague or empty descriptions with code changes affecting different concerns (tangled commits). However, our studies are exploratory in nature, with a clearly defined scope; we, therefore, do not claim that our results are generalizable, but rather make an existential argument that it is possible to detect design tradeoffs using refactorings as an indicator.

Another concern is the reproducibility of our study. To ensure that our findings are reproducible, we explicitly documented each step of our study, using existing, publicly available data and tools and published our custom-developed tool MetricHistory and associated scripts as open source.

## 4.5. Lessons Learned

In this Chapter, we presented groundwork to validate our initial assumption and answer the RQs 1, 2, and 3 presented in Section 1.3.

In Section 4.1, we explored the relationship between design, internal quality metrics, and RCs in the project JFreeChart. We found that RCs can be classified into four scenarios. Each scenario captures a specific pattern of changes. Moreover, we found that scenarios containing tradeoffs between different quality metrics were more susceptible to contain design related changes compared to the first and second scenario, validating our assumption and sketching the beginning of the answers to RQs 1, 2, and 3.

In Section 4.2, we conducted a large scale quantitative study on thirteen projects. We refine our classification into a two-dimensional taxonomy, providing an answer for RQ2. Then we observe the metric fluctuations in various development contexts. We observed a noticeable effect on metric fluctuations depending on the environmental context answering RQ3. Additionally, we observed that about half of the tradeoffs between coupling, complexity, inheritance, and cohesion captured in our approach appear in commits containing RC. This finding, connected with the results of the study presented in Section 4.1 provides further evidence towards RQ1. The conjunction demonstrates that refactorings contain valuable design changes and that metric fluctuations, particularly tradeoffs, have a certain correlation with design related changes.

Finally, we conduct a third study in Section 4.3 where we conduct a qualitative analysis over a sample of the projects to understand the extent to which tradeoffs are indicators of design related changes and intent in the context of refactoring. The results show that a measurable correlation exists, thus completing our answer to RQ1 and provides a compelling foundation for the study presented in Section 5.

# Chapter 5

# Predicting SOLID Principles

In this chapter, we aim to determine if metric fluctuations can be used as an indicator for detecting design changes (RQ4). Specifically, using the knowledge acquired in the exploratory studies from Chapter 4, we want to use metric fluctuations to detect the applications of a known set of design principles, namely the SOLID design principles (see Section 2.2).

To answer this question, we create a predictive model and measure its capacity to predict if a commit contains the application of a design principle and compare it to the ground truth. The ground truth is determined by an oracle that we created by manually analyzing commits and annotating whether they contained applications of SOLID principles.

## 5.1. Building an oracle

We annotate a sample of commits from the projects presented in Chapter 3. We sampled RCs from the entire project commit history of each project. The sampling size for each project is determined using a 95% confidence interval and a 10% margin of error [**75**]. We tagged each RC to indicate which of the five design principles were detected, and if the commit is tangled. Additionally, we assign the project an overall "commit hygiene" score to qualify the ease of readability and understandability of the commit messages and source code changes.

### 5.1.1. Protocol

To annotate the set, we adapted the protocol of 4.1 for RC with SOLID principles and was performed by the same two researchers. The consolidation phase, in this case, involved the establishment of heuristics for deciding which SOLID principle(s) influenced design changes in each RC. For instance, the redistribution of class members through Move Method, Move Attribute, Pull Up Method refactorings that improve class cohesion are typical applications of SRP. A typical indication for the application of OCP is the replacement of class dependencies on concrete implementations with abstract classes or interfaces. The introduction of Template Method and Strategy design patterns is another hint for the application of OCP. Moreover, splitting a crowded[1] interface to simpler ones denotes the application of ISP. Finally, since both OCP and DIP require classes to *depend on abstractions*, we distinguished the application of DIP on the basis of "ownership inversion" of interfaces. Ownership inversion requires that an interface is packaged in the same module with the client component that uses it. On the other hand, interface implementations are packaged in external modules that depend on the client module [**55**]. Regarding the application of LSP, we searched for cases where the design of subclasses changes for conformance to the contract of the superclass, e.g., narrowing down their public interface to match that of their parent, fixing pre/postcondition violations in concrete overriding. A transcription of the complete guidelines is available in Appendix B.

In order to gain more insight about the nature of the commits we were evaluating, we annotated two additional aspects. First, for each commit, we flag to '1' if the commit is tangled and '0' if the commit isn't tangled (See Section 2.3). Second, at the end of a project's analysis, we also assign it an overall "commit hygiene" score going from *Poor*, *Fair*, *Good*, *High*. Commits dedicated to a unique modification in the source code (e.g., adding a feature, fixing typos, refactoring an aspect of a class) and showcasing a clear message are signs of high commit hygiene [**8**] while commits bundling many changes together or described vaguely are signs of low commit hygiene because it is hard to understand what changed in the commit and defeats parts of the VCSs' purpose.

### 5.1.2. Dataset

Overall, we manually analyzed a total 928 commits from 11 projects: Ant, ArgoUML, Dagger2, Hibernate ORM, jEdit, Jena, JMeter, JUnit4, OkHttp, Retrofit, RxJava. The analysis was done by two reviewers independently after a pilot run on JFreeChart used to synchronize the protocol and clear misunderstandings. The number of RC for each project,

[1]Also known sometimes as a "fat" interface in the terminology used by M. Fowler [**22**]

**Tab. 5.1.** Summary of the collected dataset per project.

| Project | Sample size | Hygiene | Tangled | SRP | OCP | LSP | ISP | DIP |
|---|---|---|---|---|---|---|---|---|
| Ant | 91 | **High** | 25% | 26 | 9 | 8 | 3 | 2 |
| ArgoUML | **92** | Good | 42% | 26 | 9 | 5 | 2 | 3 |
| Dagger2 | 83 | Good | 16% | 26 | 7 | 4 | 1 | 1 |
| Hibernate ORM | 91 | Fair | 21% | 21 | 15 | 3 | 2 | 1 |
| jEdit | 90 | **Poor** | **70**% | 41 | 14 | 7 | 4 | 4 |
| Jena | 89 | Good | 33% | 21 | 10 | 1 | 0 | 1 |
| JFreeChart | 60 | Good | 40% | 13 | 5 | 1 | 0 | 0 |
| JMeter | 90 | Fair | 32% | 19 | 5 | 2 | 0 | 0 |
| JUnit4 | 75 | Fair | 56% | 29 | 8 | 4 | 0 | 4 |
| OkHttp | 80 | Good | 33% | 30 | 9 | 4 | 0 | 1 |
| Retrofit | **64** | Good | **9**% | 19 | 8 | 2 | 1 | 5 |
| RxJava | 83 | Good | 28% | 19 | 7 | 5 | 2 | 1 |

its associated commit hygiene and percentage of tangled commits, are presented in Tab. 5.1 as well as the number of occurrences for each design principle.

Among all projects, we found 334 SOLID commits and 594 NON-SOLID commits. It's a ratio of 1:1.78 for SOLID over NON-SOLID commits. This result is reasonable since we established in Chapter 4.2 that RCs are more likely to contain design related changes. In Fig. 5.1, we observe approximately the same ratio across each project. A notable outlier is the jEdit project which has almost the same number of commits in each category. Rather than being a sign of superior oriented object design or a systematic application of refactorings we believe this result is caused by the commit hygiene of the project. Indeed, this project was rated with a *Poor* commit hygiene, the developers often bundling many unrelated changes together in large commits as demonstrated by a high percentage of tangled commits (70%). Thus, a commit is more likely to contain design changes than a project with more commits. However, this means that recovering meaningful design information from the diff in jEdit is also harder than in projects with a better commit hygiene because the changes are hidden by other changes in the same commit.

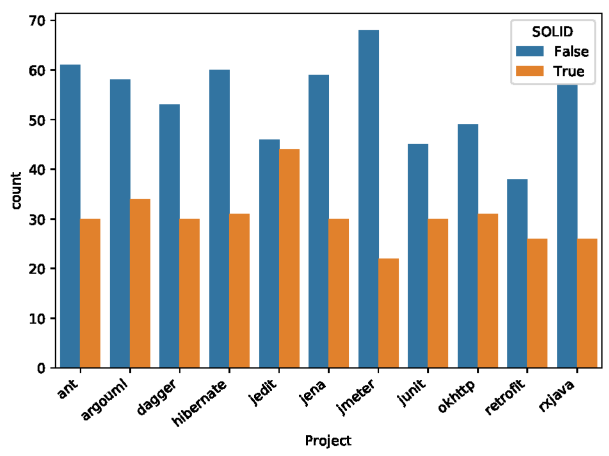


**Fig. 5.1.** The distribution of SOLID/NON-SOLID commits for each project.

Regarding the distribution of the type of SOLID principles in Tab. 5.1, we can see that SRP instances are more prevalent than any other type. By decreasing order, we found 290 instances of SRP, 106 instances of OCP, 46 instances of LSP, 23 instances of DIP, and 15 instances of ISP. Even though DIP and LSP applications are hard to find in the source code, we haven't been confronted to many situations were this kind of design principle were applied. Regarding DIP especially, we observed that dependency inversion was barely used even though it is considered to have major benefits for the maintainability of the source code. Whether this apparent disparity comes from a conscious decision from developers or pure happenstance is unclear. Design principles like SRP are notably easier to understand and implement than DIP. However, the previous argument do not explain the low number of ISP instances found. Applications of ISP are relatively easy to spot and easy to implement and understand for developers. We believe that this design principle is too specific too be found in large quantities as a general principle such as SRP.

Concerning the commit hygiene scores, most of the projects we studied were decent (*Good*, *High*). Since the hygiene score is determined subjectively by the reviewer, it is hard to definitely rank the projects but it is intended as an estimate of what to expect in terms of comprehension difficulty if a researcher or developer was to read the diffs. Moreover, this also means we cannot establish a formal correlation between the hygiene score and the percentage of tangled commits for the projects studied. At first glance, it seems that there is no obvious correlation. We believe it's due to the fact that not all tangled commits are equal. There is multiple ways for a commit to become tangled and they are not equal it terms of added complexity when trying to understand the code changes they incur.

## 5.2. Approach

To build our model, we use a decision tree approach from the family of supervised learning. We use decision trees for their capacity to model complex phenomena and their explicability. Moreover, decision trees are great at filtering relevant features which will inform us on which metrics are instrumental, among the 52 we extracted via SourceMeter that are using to train this model, in order to build a first intuition towards a theory of change patterns.

We train the model using the fluctuations for each commit as features and the oracle data collected previously as target value. The target value is a binary value indicating that either a commit contains an application of SOLID principles or it doesn't. We are not trying to detect which specific principle was applied at this time as we have a low class representation for LSP, ISP, DIP. We use the metric fluctuations in RCs because they have a high density of design related changes compared to regular commits as found in Section 4.2. The advantage is that we are more likely to have relevant examples for training and more balanced training sets. Before the model is trained, we run hyperparameter optimization on the training set with cross-validation to select the best hyperparameters for the decision tree. To mitigate the imbalance in SOLID and NON-SOLID commits, we use a weighted training and evaluation method which mitigates the possible overfitting caused by unbalanced datasets [**23**].

Since we are interested in a general model capable of predicting the application of design principles for commits of any project, we have to evaluate its performance on new projects. Simply splitting the available data into training and testing sets would not be an accurate evaluation of performance. Thus, we reserve the data for one arbitrary project for testing (acting as the "new project") and use the remaining projects' data for the training. This process is repeated such that each project is reserved in turn and tested as the "new project". The results for each experiment are then averaged to give the final performance of our approach. This procedure guarantees that new commits are never seen during hyperparameter optimization or training and make the most of our available dataset.

## 5.3. Results

We evaluate the performance of our models on several metrics. We use accuracy, the Area Under The Curve (AUC) of the Receiver Operating Characteristics (ROC), often abbreviated as Area Under the Receiver Operating Characteristics (AUROC) [**62**], and the F1 Score [**84**]. The results are shown in Tab. 5.2.

**Tab. 5.2.** Average performance of the approach with respective standard deviations.

| Accuracy | AUROC | F1 | F1 SOLID | F1 NON-SOLID |
|---|---|---|---|---|
| 65.9% (±5) | 68.8% (±7) | 66.4% (±7) | 58.8% (±8) | 69.8% (±1) |

We obtain an average weighted accuracy of 65.9% which indicates that our model is somewhat capable of detecting the application of SOLID principles. However, this does not inform us of the capacity of the model to distinguish between our two class of data: SOLID and NON-SOLID. For this, we use AUROC. A value of 50% signifies the model has no class separation capacity, similar to a random classification. We use this value as our baseline. We obtained an average of 68.8% for AUROC for our model, confirming the capacity of the model of making adequate predictions most of the time. Interpreted, it means our model has about 68.8% of chance to predict the right class. The F1 score is another measure of prediction performance based on the precision (number of selected items that are relevant) and recall (number of relevant items that are selected). We obtain a value of 66.4% which indicate again a certain capacity of prediction for the model.

Finally, F1 SOLID and F1 NON-SOLID inform us on the capacity of the model to make predictions for each class. We see that in average, our models had an easier time guessing RC containing no application of SOLID principles as suggested by the higher F1 score for NON-SOLID than SOLID. Additionally, we can see that the NON-SOLID predictions are more reliably learned with a low standard deviation of 1%. SOLID predictions, however, suffer from a high standard deviation (8%) relatively to the average F1 SOLID score.

Next, we want to understand what metrics are most useful for the classification. To do this, we rank the metrics usage by the classifiers in Tab. 5.3. For each model, we count the metrics in the first 3 levels starting from the root that appear at least twice among all projects. An exact description of each metric is available on the official documentation of SourceMeter [**3**].

**Tab. 5.3.** Number of occurrences for recurring metric in the three first levels of the decision trees.

| Metric | Abbreviation | Occurrences |
|---|---|---|
| Number of Local Attributes | NLA | 8 |
| Total Number of Methods | TNM | 7 |
| Coupling Between Objects | CBO | 6 |
| Total Number of Private Methods | TNPM | 6 |
| Number of Local Public Attributes | NLPA | 5 |
| Total Number of Local Attributes | TNLA | 4 |
| Number of Incoming Invocations | NII | 4 |
| Total Logical Lines of Code | TLLOC | 4 |
| Total Number of Local Getters | TNLG | 3 |
| Number of Parents | NOP | 3 |
| Total Number of Attributes | TNA | 2 |
| Total Number of Statements | TNOS | 2 |
| Documentation Lines of Code | DLOC | 2 |
| Number of Statements | NOS | 2 |
| Public Documented API | PDA | 2 |
| Number of Attributes | NA | 2 |

We observe that the most useful metrics are NLA and TNM. They are almost always present as one of the most decisive variables. Moreover, we found that NLA was the root metric in 8 out of 11 times while TNM was in the one 3 remaining. This is a surprising result for a size metric. We also see that most of the top metrics are concerned with either attributes or methods. Only a handful are concerned with classes or documentation which is also counter intuitive since we predicted the application of SOLID principles, inherently operating at a higher granularity than attributes and methods. Overall, 11 of the metrics presented are measuring size. 2 metrics (CBO, NII) are measuring coupling. 2 metrics (DLOC, PDA) are measuring documentation, and a single metric is measuring inheritance.

There is no metric measuring cohesion or complexity. A possible explanation may be that other types of metrics are not precise enough given the cleanliness of the data (tangled changes) so the model defaults to using size metrics as its best proxy. Another aspect to take into account is the redundancy between some metrics (e.g., there are five metrics concerning attributes). It would be interesting to force the model to use only one per type of object and see then which is the most relevant.

## 5.4. Preliminary Evaluation of Usefulness

Our objective is not only to provide a model that can generalize to other projects, but whose predictions are actually helping the developers. In addition to the performance validation described in Section 5.2, we investigate whether our model is able to give predictions that are relevant to developers for any commit (i.e., not only RCs) for new projects. In order to do this, we took a dogfooding approach where we apply our approach presented in this chapter on our own projects [**30**].

### 5.4.1. Protocol

T. Schweizer and V. Zafeiris will evaluate the predictions from the model on their respective projects MetricHistory (356 commits and 3977 LLOCs), and a proprietary project that measure software quality in governmental projects, anonymized $Q$. In parallel, we compute the metric fluctuations for the entire revision history of both projects using the procedure described in Chapter 3. Then, we train a general model using the approach and the data presented in the Sections 5.1 and 5.2 sections. Afterwards, we run the metric fluctuations of MetricHistory and $Q$ through the classifier to obtain the predictions of the applications SOLID principles, and we selected 10 commits classified as SOLID and 10 commits classified as NON-SOLID for each project. Finally, both developers described the extent to which they agreed, or disagreed with the prediction of the classifier for the sample of 20 commits from their respective projects:

- if a commit was tagged as SOLID, whether they agreed that this commit contained changes related to design
- if the commit was tagged as NON-SOLID, whether they agreed that this commit did not contain changes related to design

We use a Likert scale [**48**] with the options *Disagree*, *Somewhat disagree*, *Neutral*, *Somewhat agree*, and *Agree* to qualify their agreement. *Disagree* signifies a total disagreement while *Agree* signifies a total agreement with the question presented above.

### 5.4.2. Results

For each project, we count 1) the overall agreement, composed of *Agree* and *Somewhat agree*; 2) the overall disagreement, composed of *Disagree* and *Somewhat disagree*; 3) the SOLID agreement, composed of *Agree* and *Somewhat agree* responses on commits predicted as SOLID; 4) the NON-SOLID agreement, composed of *Agree* and *Somewhat agree* responses on commits predicted as NON-SOLID. The results are presented in Tab. 5.4. We also show the decision tree model in Fig. 5.2.

**Tab. 5.4.** Tally of the developers opinions for projects MetricHistory et $Q$.

| | MetricHistory | $Q$ |
|---|---|---|
| Overall agreement | 13 (65%) | 16 (84%) |
| Overall disagreement | 7 (35%) | 3 (16%) |
| SOLID agreement | 4 (40%) | 7 (70%) |
| NON-SOLID agreement | 9 (90%) | 9 (90%) |

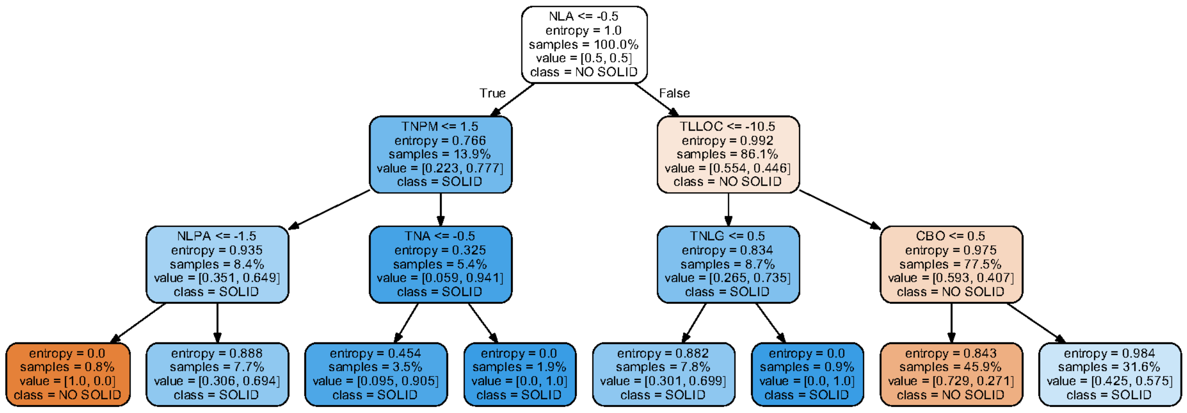


**Fig. 5.2.** The decision tree model used for the evaluation.

Globally, $Q$ contains the only instance where the developers had no opinion about a prediction. Looking at the overall agreement and disagreement, we can see that developers were mostly agreeing with the results of the classifier as demonstrated by a combined total of 29 $(13 + 16)$ positive opinions versus 10 $(7 + 3)$ negative opinions. This is an encouraging finding that signifies that the classifier was relevant in its task and was able to give useful results in practice.

Looking at the SOLID agreement, we observe that the model had equivocal results. In MetricHistory slightly less than half of the commits classified as containing SOLID principles actually contained relevant design knowledge for the developer, while in $Q$, more than half of the commits classified as containing SOLID principle were actually useful for the developer. Given our limited sample of projects, this can indicate that the classifier performs differently depending on the project but also that we need to gather more opinions for a given commit from multiple developers as their opinion is, definition, subjective. However, the NON-SOLID agreement seems to be converging for both developers. Indeed they agreed with the prediction of the classifier for almost all commits classified as NON-SOLID. This result aligns with the F1 scores obtained previously which indicated that our model performed better in detecting NON-SOLID instances than SOLID instances.

We provide the model as well as the script to generate it in a replication package on Zenodo [**67**].

## 5.5. Threats to validity

In Section 4.4, we described the threats to validity related to the mining of the fluctuations in metrics. In this section, we will address the threats to validity affecting the creation of the annotated dataset, and the user study.

The process of detecting design intent and applications of SOLID principles is subject to researcher bias which constitutes a threat to internal validity. To mitigate this, a calibration analysis was exercised independently by two reviewers on all the RCs of JFreeChart. Then we proceeded to resolve any conflicts by consensus and and created a common annotation protocol as a result. The full protocol is available in Appendix B.

The usefulness study is also threatened by researcher bias since the developers are also the researchers of the study. However, the objective of this usefulness study was not to prove the general relevance and usefulness of our model but to verify that the predictions of the classifier were sound and could be transposed to real projects.

## 5.6. Lessons Learned

In this chapter, we put the observations from the exploratory studies in Chapter 4 into practice. We were able to demonstrate that metric fluctuations can be used as indicators of design knowledge, SOLID design principles in our case (RQ4). The results we obtained are encouraging despite their limitations, and given that we made certain simplifications to our approach such as using a naïve aggregation process and considering commits containing multiple sets of unrelated changes. In Chapter 7, we propose certain possible avenues for improvement that could result in better prediction performance, especially for detecting positive instances of SOLID applications.

During the process of creating the oracle, we noticed that it was sometimes possible to guess when a functionality was moved around using only the changes in lines of code. Most of the time, it's only a few lines that change across files: the developer will wrap an if condition around a group of statements or some functionality will be tweaked; radical changes are rather an exception. Thus, the displacement of a concept to other files in the project will stand out from the usual editing pattern described above. We think it's a similar process that made our model privilege size metric changes than the coupling, cohesion, and inheritance metrics we expected. We believe that when we will refine our methodology, we will be able to tap into the metrics that are eluding the model currently. Moreover, this is also a sign that reinforces the intuition presented in Chapter 1 that fluctuations in metrics can be used to trace macroscopic changes relevant to developers.

# Chapter 6

# Related work

Metrics have played an important role in evaluating the quality of a software system and in guiding its design and evolution. This role is summarized very well by Stroggylos et al. [**74**], who present a set of research works that have used metrics for these tasks especially in the context of refactoring. The authors also present a study where, similarly to our work, they measure software quality metrics before and after refactorings for a set of object-oriented systems. According to their findings, the impact on metrics depends on the subject system, the type of refactoring and in some cases on the tool used to measure the metrics. Nevertheless, the impact is not always positive, as one would expect. This has motivated our study and definition of tradeoffs, in order to correlate metrics and refactoring activity with design intent that possibly justifies any potential deterioration in quality metrics. In our study, we automated the refactoring detection, allowing us to measure the effect on the design of more projects. We were able to confirm that refactoring activities does not necessarily lead to improvements in quality in 13 projects compared to their four original projects. Additionally, we provide an intuitive and scalable characterization of commits by classifying them in scenarios and then in a refined taxonomy.

Each type of refactoring may affect multiple metrics and not always in the same direction. Researchers have explored this complex impact to detect design problems known as code smells. Marinescu defined thresholds on a number of metrics and then combined those using AND/OR operators in rules called detection strategies [**53**]. A detection strategy could identify an instance of a design anomaly, which could orthogonally be fixed by a corresponding refactoring. A very similar approach, using metrics and thresholds, was followed by Munro et al. [**60**]. Although, in principle, metric tradeoffs could be captured in detection rules, Tsantalis et al. went one step further and defined a new metric to capture a tradeoff [**79**]. Since, coupling and cohesion metrics can often be impacted in opposite directions during refactoring [**74**], Tsantalis and Chatzigeorgiou defined a new metric, *Entity Placement*, that combines coupling and cohesion. Quality assessment using Entity Placement is supposed to give more global results with respect to detection and improvement after refactoring. Kádár et al. and Hegedüs et al. focused exclusively on the relation between metrics and maintainability in-between releases [**38, 31**]. A cyclic relation was found, where low maintainability leads to extended refactoring activity, which in turn increases the quality of the system.

The activity of refactoring and its relation to design has been extensively studied. Chávez et al. performed a large-scale study to understand how refactoring affects internal quality attributes at a metric level [**13**]. In contrast, our study aims at exploring the specific role of refactoring on internal qualities when metrics embody a tradeoff and how it relates to design. Cedrim et al. investigated the extent to which developers are successful at removing code smells while refactoring [**10**]. Soetens et al. analyzed the effects of refactorings on the code's complexity [**71**]. Tsantalis et al. investigated refactorings across 3 projects and examined the relationship between refactoring activity, test code and release time [**80**]. They found that refactoring activity is increased before release and it's mostly targeted at resolving code smells. Compared to the study presented in this paper, their study was not guided by metrics, nor did it involve extensive qualitative analysis and, finally, it did not discuss more major design decisions as part of the intent.

Recovery of design decisions has been studied from an architectural perspective. Jansen et al. proposed a methodology for recovering architectural design decisions across releases of a software system [**37**]. The methodology provides a systematic procedure for keeping up-to-date the architecture documentation and prescribes the steps that the software architect must follow in order to identify and document architectural design decisions across releases. A fully automated technique for the recovery of architectural design decisions has been, recently, proposed by Shahbazian et al.[**68**]. The technique extracts architectural changes from the version history of a project and maps them to relevant issues from the project's issue tracker. Each disconnected subgraph of the resulting graph corresponds to an architectural decision. The recovered decisions are relevant to structural changes in system components, applied across successive releases.Our method focuses on decisions affecting detailed design and concern the structure of classes and the distribution of state and behavior among them.

Moreover, decision recovery takes place at the revision level and is guided by metrics' fluctuations and fine-grained changes due to refactorings. Besides, we employ issue tracker information for manual cross-checking of design decisions as well as commit messages and source code comments. Their big picture is very similar to ours. However, there is considerable differences in the approaches. Our main goal is to find design tradeoffs. In our study, the tradeoffs emerge from the metric fluctuations between versions. The design detection is a component of our process that is done manually. Another big difference with their study is the size. They analyze fewer version than us but the projects they choose have a high density of issues that can be mapped to commits. Our method focuses on decisions affecting the lower level design and which concern the structure of classes and the distribution of state and behavior among them.

# Chapter 7

# Conclusion

## 7.1. Summary

Our vision is to help developers better understand the code artifacts they work with. With this research, we took the first step by studying the fluctuations in quality metrics in projects' revision histories. By characterizing the metric fluctuations we aim to better understand change, especially design, and to facilitate the detection of change patterns that can be relevant as documentation to developers or other stakeholders.

Given the foundational nature of our research, we first presented an qualitative exploration of JFreeChart. Then, we refine our comprehension of the domain in a second study where we conduct a quantitative study of the entire revision history of thirteen projects that characterizes metric fluctuations categories and observing how they change in different contexts. In a third step, we conduct a qualitative study on a sample of commits to understand if internal quality metrics in the context of refactoring are correlated with design intent from developers. Finally, having tested our hypothesis and built a significant dataset, we built a model to detect applications of SOLID principles in commits.

We observed that source code changes containing refactorings affects internal quality metrics and design in a number of ways. Refactoring induced changes are more likely to improve or deteriorate the internal quality of a software than commits without refactorings. Moreover, RCs containing certain metric tradeoffs were found to be relevant indicators for important design quality tradeoffs, likely embodying the design intent of the author (RQ1). Secondly, we created a robust taxonomy to characterize metric fluctuations on two, but not limited to, dimensions enabling researchers to navigate through vast number of commits and select relevant change patterns (RQ2). Thirdly, we found that there is a dependency between metric fluctuations and development context. Metric fluctuations will change based on the context, giving insight on the effect of a development activity like refactoring or the type of changes between test code and business logic code (RQ3). Finally, we completed this first step by demonstrating that metric fluctuations can give insight about a software's design by building a model using the fluctuations of 52 metrics to detect the application of SOLID design principles with a 68.8% class prediction accuracy and an average F1 score of 66.4% (RQ4). In conclusion, through a assortment of qualitative and quantitative exploratory studies, we were able to demonstrate that metric fluctuations in RCs can be used to gain insights about the design evolution of the source code.

## 7.2. Limitations & Future Work

In this section, we discuss the next steps to be addressed in order to generate historical based design meta-data for each artifact. We envision this meta-data to automatically document design concerns and facilitate code comprehension tasks for developers, enabling them to make reliable and informed development decisions.

### 7.2.1. Technical Improvements

The most straightforward future work is to improve the data acquisition pipeline in making it easier to use and more flexible for researchers.

For a study about design and refactoring, several factors can be important in selecting a project. Depending on the research questions and what one is looking for, important factors may include programming language, architecture style (object-oriented or other), history length (i.e., number of revisions or commits), number of developers and contributors and others. Thankfully, most VCS service providers, like GitHub [**34**], Bitbucket [**4**], GitLab [**35**], which contain a large number of open source software systems of great variety, already contain this metadata for each project. Therefore, we can integrate a repository crawler like Sourcerer [**49**] or develop a customizable one, where we can specify the criteria and fetch the repositories to be analyzed.

Despite the vast use of Git in software development nowadays, some projects, particularly older ones, are often hosted on different VCSs such as Subversion [**21**], Mercurial [**51**], or

CVS [**77**]. We want to support these systems in MetricHistory so that researchers have access to a larger pool of projects and enables them to see if VCSs have an impact on the development process.

Another current limitation is the time it takes to extract the metric for each commit of a project, especially ones with an large number of lines of codes or projects with a long version history. In our research, we used Akka [**47**] to distribute the workload on multiple computers. This setup, while very effective, requires extra time to arrange and learning how the framework works. Optimally, we want a seamless implementation of distributed computing behind MetricHistory where the researcher wouldn't have to think about it, saving him a lot of time in learning and troubleshooting.

Staying in the domain of computation, we found that we could process a high number of commits in large projects if instead of computing from scratch the metrics for each version, we base the metric computation on the previous version plus the changes, effectively doing incremental static analysis between versions. This is not a straightforward task, particularly for metrics measuring dependencies such as incoming and outgoing invocations.

Finally, metrics are currently produced and stored in Comma Separated Value (CSV) files. To accommodate a large volume of measurements, MetricHistory can already split the data into one file of metrics per commit, avoiding the generation of very large files (i.e., almost a hundred gigabytes in certain projects) and the subsequent problems faced when trying to work with them such as limited RAM and parsing time. These concerns can entirely be avoided if we integrate a database as a core part of our methodology and process. Measurements would directly be saved in the database and the post processing would query the database for only what it needs. This solution would reduce storage needs of researcher and improve the speed of saving new data and accessing existing data. Moreover, this configuration comes with a significant advantage: it scales. Indeed, it would be straightforward to host the database in the cloud or in a server farm and add as many resources necessary. This last benefit brings another one. It enables us to open our dataset to collaboration to other researchers by providing the public address of the database instead of giving a static copy of our dataset. This way, researchers can access the database for their own studies and even enrich the database.

### 7.2.2. Methodology Improvements

Regarding methodology, we identified multiple areas that can be improved. First of all, regarding metrics, we intend to expand our analysis to include more quality metrics, as well as other types of static analysis (e.g., bugs, anti-patterns, code smells). Additionally, we want to look at the effect of granularity on fluctuations measurements. In these studies, we only look at the commit level. We believe interesting change patterns can emerge on software classes or packages through time. Finally, we want to also include the relative changes in metrics between two versions. Currently, we only use absolute values which we believe loses information.

The quality of the version history is a critical aspect for our method. By being confronted to a diversity of version histories, we were able to confirm that the hygiene of the commits in a project has a direct impact on the difficulty and time necessary to understand the project and source code changes. Specifically, we noticed that commits containing tangled changes are problematic. These commits are harder to understand for researchers and developers alike. Moreover, they also introduce non-negligible noise that we believe is detrimental to our approach. To solve this issue we want to find a method to untangle unrelated sets of changes inside one commit leveraging existing work on tangled commits [**2**] and software traceability [**73, 45**].

Another critical aspect is our aggregation process. Currently, we use a simple approach where we sum the changes of metrics for each class for each metric. As discussed previously, this technique has several drawbacks. We want to refine this aggregation process by working with distributions of changes for a given metrics instead of collapsing all the change points into one value. This distribution would enable more complex analysis and use advanced statistical tools to identify how that changes are distributed at the level of a commit.

Finally, we want to compare the decision tree model we used detect SOLID principles with other supervised learning approaches such as linear models, support vector machines, stochastic gradient descent, nearest neighbors, Bayesian approaches, and ensemble methods [**23**].

### 7.2.3. Indicators of design

In future work, we want to investigate whether we can use alternative sources such as self-admitted technical debt, code documentation, and bugs as indicators of design knowledge. Additionally, we want to encompass a larger set of design changes and intent than a subset of architectural concerns. We want to include as indicators additional architectural changes but also factors such as semantic changes, frameworks, and language choice. The rationale is that the process of software development and thus designing software is never done in a vacuum but is tributary to the context in which its developed. Certain programming languages and paradigms prefer certain code structures while others privilege something else. This is also true for frameworks which often require particular ways of programming, forcing the developer to write in a certain way. Semantic changes, such as renames, are also a powerful tool to understand the design choices among of group of software artifacts since, in a world of abstract concepts, names and their relations form the cognitive fabric of an application; they give meaning to the arrangements of code developers come up with to satisfy the stakeholders requirements.

### 7.2.4. Design Intents and Cross-referencing

The manual identification of design intent was possibly the most complicated step of our process and the one that required extensive manual effort. The reason for this is because refactorings, design decisions and generally the intent of developers is not always explicitly expressed in comments or documentation. For JFreeChart, the artifact that conveyed the most information was the changelog. The changelog was Javadoc comments that preceded one or more classes within a commit (usually classes that had the most significant changes) and described in natural language how the classes were changed. Conversely, commit comments did not contain much information about design intent or something more high-level other than the change itself. In order to understand the intent, we studied the commit comments, the changelog and the source code itself, and we went back in the project's history to understand more about the changes and the evolution of the metrics. The automation of this step would require significant effort. The first idea is to employ Natural Language Processing (NLP) to mine all textual data of the project (e.g., mails, commit comments, source code comments, changelogs) and look for specific textual patterns pertaining to design and design maintenance. Next, we will use time-series clustering to find relationships between current and previous changes to track design decisions through the project's history. Finally, we will use association rules to complete the cross-reference step between the classified commit according to metrics and the design decisions as identified before. It is not certain that manual effort will be completely eradicated even after the use of these machine learning techniques, but the goal is to minimize it as much as possible and increase the accuracy of the analysis.

# Bibliography

[1] Jehad Al Dallal and Anas Abdin. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Trans. Softw. Eng.*, 44(1):44–69, January 2018.

[2] Ryo Arima, Yoshiki Higo, and Shinji Kusumoto. A study on inappropriately partitioned commits: How much and what kinds of ip commits in java projects? In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 336–340, New York, NY, USA, 2018. ACM.

[3] Frontend Art. Sourcemeter java documentation. `https://www.sourcemeter.com/resources/java/`. [Online; accessed 2020-01-10].

[4] Atlassian. Bitbucket. `https://bitbucket.org/`. [Online; accessed 2019-01-10].

[5] Thomas Ball, J. Kim, Adam Porter, and Harvey Siy. If your version control system could talk ... 10 1997.

[6] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on software engineering*, 28(1):4–17, 2002.

[7] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. *Recommending Refactoring Operations in Large Software Systems*, pages 387–419. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[8] Chris Beams. How to write a git commit message. `https://chris.beams.io/posts/git-commit/`. [Online; accessed 2019-01-10].

[9] Grady Booch. *Object oriented analysis & design with application.* Pearson Education India, 2006.

[10] Diego Cedrim, Leonardo Sousa, Alessandro Garcia, and Rohit Gheyi. Does refactoring improve software structural quality? a longitudinal study of 25 projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*, SBES '16, pages 73–82, New York, NY, USA, 2016. ACM.

[11] Scott Chacon and Ben Straub. *Pro Git.* Apress, Berkely, CA, USA, 2nd edition, 2014.

[12] Ned Chapin, Joanne E Hale, Khaled Md Khan, Juan F Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, 13(1):3–30, 2001.

[13] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. How does refactoring affect internal quality attributes?: A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, SBES'17, pages 74–83, New York, NY, USA, 2017. ACM.

[14] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[15] MSR Steering Comitee. Conference on mining software repositories. `http://www.msrconf.org/`. [Online; accessed 2020-01-10].

[16] Git community. Git. `https://git-scm.com/`. [Online; accessed 2019-01-10].

[17] Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Findings from github: methods, datasets and limitations. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 137–141. IEEE, 2016.

[18] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017.

[19] Mário André de F. Farias, Renato Novais, Methanias Colaço Júnior, Luís Paulo da Silva Carvalho, Manoel Mendonça, and Rodrigo Oliveira Spínola. A systematic mapping study on mining software repositories. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1472–1479, 2016.

[20] R. Ferenc, L. Langó, I. Siket, T. Gyimóthy, and T. Bakota. Source meter sonar qube plug-in. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 77–82, Sept 2014.

[21] Apache Software Foundation. Subversion. `https://subversion.apache.org/`. [Online; accessed 2019-01-10].

[22] Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley Professional, Boston, MA, USA, 2018.

[23] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[24] Cristina Gacek and Budi Arief. The many meanings of open source. *IEEE software*, 21(1):34–40, 2004.

[25] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 190–198, Nov 1998.

[26] David Gilbert. The jfreechart class library. *Developer Guide. Object Refinery*, 7, 2002.

[27] GitHub. Github importer. `http://bit.ly/2ZNqcT0`. [Online; accessed 2019-04-24].

[28] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21, 2012.

[29] Yann-Gaël Guéhéneuc. Ptidej: A flexible reverse engineering tool suite. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 529–530. IEEE, 2007.

[30] W. Harrison. Eating your own dog food. *IEEE Software*, 23(3):5–7, May 2006.

[31] Péter Hegedüs, István Kádár, Rudolf Ferenc, and Tibor Gyimóthy. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information & Software Technology*, 95:313–327, 2018.

[32] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 121–130, Piscataway, NJ, USA, 2013. IEEE Press.

[33] Alan Hevner and Samir Chatterjee. *A Science of Design for Software-Intensive Systems*, volume 47, pages 63–77. 06 2010.

[34] GitHub Inc. Github. `https://github.com/`. [Online; accessed 2019-01-10].

[35] GitLab Inc. Gitlab. `https://gitlab.com/`. [Online; accessed 2019-01-10].

[36] Gradle Inc. Gradle. `https://gradle.org/`. [Online; accessed 2019-01-10].

[37] Anton Jansen, Jan Bosch, and Paris Avgeriou. Documenting after the fact: Recovering architectural design decisions. *Journal of Systems and Software*, 81(4):536 – 557, 2008. Selected papers from the 10th Conference on Software Maintenance and Reengineering (CSMR 2006).

[38] I. Kádár, P. Hegedus, R. Ferenc, and T. Gyimóthy. A code refactoring dataset and its assessment regarding software maintainability. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 599–603, March 2016.

[39] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, 19(2):77–131, 2007.

[40] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101, 2014.

[41] Barbara A. Kitchenham. *Software Metrics: Measurement for Software Process Improvement*. Blackwell Publishers, Inc., Cambridge, MA, USA, 1996.

[42] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, Nov 2012.

[43] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, 33, 1977.

[44] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[45] Yi Li, Chenguang Zhu, Milos Gligoric, Julia Rubin, and Marsha Chechik. Precise semantic history slicing through dynamic delta refinement. *Automated Software Engineering*, 26(4):757–793, Dec 2019.

[46] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, June 1978.

[47] Lightbend. Akka. `https://akka.io/`. [Online; accessed 2019-04-08].

[48] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[49] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2009.

[50] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

[51] Matt Mackall. Mercurial. `https://www.mercurial-scm.org/`. [Online; accessed 2019-01-10].

[52] L. MacLeod, M. Greiler, M. Storey, C. Bird, and J. Czerwonka. Code reviewing in the trenches: Challenges and best practices. *IEEE Software*, 35(4):34–42, July 2018.

[53] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359. IEEE, 2004.

[54] Robert C Martin. Design principles and design patterns. *Object Mentor*, 1(34):597, 2000.

[55] Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C# (Robert C. Martin).* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[56] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[57] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz. The work life of developers: Activities, switches and perceived productivity. *IEEE Transactions on Software Engineering*, 43(12):1178–1193, Dec 2017.

[58] Bertrand Meyer. *Object-oriented software construction (2nd ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[59] Martin Michlmayr, Francis Hunt, and David Probert. Quality practices and problems in free software projects. In *Proceedings of the First International Conference on Open Source Systems*, pages 24–28, 2005.

[60] Matthew James Munro. Product metrics for automatic identification of" bad smell" design problems in java source-code. In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 15–15. IEEE, 2005.

[61] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012.

[62] Sarang Narkhede. Understanding auc - roc curve. `https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5`. [Online; accessed 2019-01-10].

[63] William F Opdyke. Refactoring object-oriented frameworks. 1992.

[64] Meilir Page-Jones. What every programmer should know about object-oriented design. *dimensions*, 227(29):252–343, 1995.

[65] Aniket Potdar and Emad Shihab. An exploratory study on self-admitted technical debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 91–100. IEEE, 2014.

[66] Martin P. Robillard. Sustainable software design. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 920–923, New York, NY, USA, 2016. ACM.

[67] Thomas Schweizer. Decision tree model and scripts. `https://doi.org/10.5281/zenodo.3722550`, March 2020.

[68] A. Shahbazian, Y. Kyu Lee, D. Le, Y. Brun, and N. Medvidovic. Recovering architectural design decisions. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 95–9509, April 2018.

[69] D. Silva and M. T. Valente. Refdiff: Detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 269–279, May 2017.

[70] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 858–870, New York, NY, USA, 2016. ACM.

[71] Quinten David Soetens and Serge Demeyer. Studying the effect of refactorings: a complexity metrics perspective. In *International Conference on the Quality of Information and Communications Technology*, pages 313–318. IEEE, 2010.

[72] S. Sothornprapakorn, S. Hayashi, and M. Saeki. Visualizing a tangled change for supporting its decomposition and commit construction. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 74–79, July 2018.

[73] George Spanoudakis and Andrea Zisman. Software traceability: a roadmap. In *Handbook Of Software Engineering And Knowledge Engineering: Vol 3: Recent Advances*, pages 395–428. World Scientific, 2005.

[74] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth International Workshop on*, pages 10–10, May 2007.

[75] Creative Research Systems. Sample size calculator. `https://www.surveysystem.com/sscalc.htm`. [Online; accessed 2019-01-10].

[76] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: An exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 51:1–51:11, New York, NY, USA, 2012. ACM.

[77] The CVS Team. Cvs. `http://savannah.nongnu.org/projects/cvs`. [Online; accessed 2019-01-10].

[78] Vassilis Zafeiris Thomas Schweizer. Metrichistory. `http://github.com/thomsch/metric-history`, 2018. [Online; accessed 2020-01-10].

[79] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.

[80] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 132–146, Riverton, NJ, USA, 2013. IBM Corp.

[81] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *40th International Conference on Software Engineering (ICSE 2018)*, Gothenburg, Sweden, May 27 - June 3 2018. IEEE.

[82] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105 – 119, 2002.

[83] Giovanni Viviani, Calahan Janik-Jones, Michalis Famelis, Xin Xia, and Gail C. Murphy. What design topics do developers discuss? In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 328–331, New York, NY, USA, 2018. ACM.

[84] Wikipedia. F1 score. `https://en.wikipedia.org/wiki/F1_score`. [Online; accessed 2019-01-10].

[85] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[86] Zhenchang Xing and Eleni Stroulia. Refactoring detection based on umldiff change-facts queries. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 263–274. IEEE Computer Society, 2006.

[87] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.

# Appendix A

## Metric History

MetricHistory is an extensible tool designed to collect and process software measurements across multiple versions of a code base. The measurement itself is modular and executed by a third party tool. The default analyzer is SourceMeter [**20**] which offers more than 52 metrics at the project, package, class, and method level.

## A.1. Features

- Automated collection of measurements for multiple version of a project.
- Computation of the difference in metrics between two version of a project.

On top of these features, MetricHistory offers a collection of optional utilities:

- Conversion of native analysis results into a easily readable reference format (RAW format).
- Retrieval of the parent versions of a list of versions.
- Exports to Mongodb database (Incubating).

## A.2. Installation

Download the latest release at `https://github.com/Thomsch/metric-history/releases`. Unzip it and run `'./bin/metric-history -V'` to check if your installation is good. This instruction should print the current version. If you plan on using SourceMeter as the analyzer, you need to install it beforehand from the vendor's website.

## A.3. Usage

### A.3.1. Running from command line

MetricHistory comes with a rich command line interface. Every command is explained in the `'help'` command (`./metric-history help`).

For example, to collect the metrics from a list of version using SourceMeter type

```
./metric-history collect versions.txt path/to/repository/ output/folder/
SOURCEMETER -e=path/to/sourcemeter/executable
```

This command analyzes the versions given in *versions.txt* of the project located in *path/-to/repository* using the analyzer 'SourceMeter'. *versions.txt* contains a list of commit ids (in case of a git project). The results are stored in `output/folder`.

### A.3.2. Using the API

You can also choose to integrate metric history to your projects by using its public API. Inspire yourself from the implementations in `org.metrichistory.cmd.*`; they all use the public API!

## A.4. Building

We use Gradle [**36**] to manage dependencies and compile the project the project.

- Run `./gradlew build` (or `./gradlew.bat ...` if you're on Windows) to build the project. Dependencies will be download automatically.
- Run `./gradlew installDist` to install the application locally.
- Run `./gradlew distZip` to create a full distribution ZIP archive including runtime libraries and OS specific scripts.

## A.5. Testing

Run `./gradlew test` to execute the tests.

# Appendix B

## Codebook

The annotation handbook presented below is copied as-is from its original document. The guidelines have been co-written with Dr. Vassilis Zafeiris who authored most of the content. This version of the codebook was used as the guidelines for the creation of the oracle presented in Chapter 5.

## B.1. Annotation guidelines

This work studies the recovery of design intent from the revision history of a software project. The focus is on the identification of revisions whose change-set introduces structural improvements to system design. The recovery of design intent from software revisions would contribute to better understanding the evolution of a project through identification of important milestones in its lifecycle. Besides retrospection, the identification of design intent in new commits could enhance the code review process by giving priority to those commits that alter design and could potentially impact the architectural conformance of the software. We intend to create a classifier that characterizes the presence or not of design intent to a given revision.

The proposed method involves the following assumptions:

- Design intent denotes a deliberate attempt to improve system design
- Since the notion of design improvement is rather abstract and may refer to a broad set of design options, we proxy it with the intent for enforcement of well established OO design principles
- We adopt the SOLID set of design principles that focus on code maintainability

- The enforcement of a design principle denotes the introduction of structural changes to existing classes so as to achieve better conformance to the given principle, e.g. Single Responsibility Principle through relocation of functionality
- Structural changes relevant to design improvement can be traced to the application of refactorings.
- The identification of structural changes in software revisions can be reduced to mining the application of refactorings.
- Design interventions may, also (on purpose or not), deteriorate design, as manifested by the violation of design principles. Tracking of violations of design principles is currently out of context of this work, but could be an interesting part of another study. However, in these cases, refactorings cannot always reveal these violations. Most probably these could be traced to additions of methods/fields/dependencies to a given class.

Our method tries to identify the design intent on the basis of more primitive features of a revision: (a) the presence of refactorings and (a) the change in four object oriented metrics. Should we use the presence of each individual refactoring as a separate feature?

### B.1.1. Classifier for SOLID principles detection in RCs

- What if the presence of each principle is correlated with specific types of refactorings (e.g. SRP with extract method, OCP with extract interface/superclass etc.)? In this case metric fluctuations will have no effect on the classifier. Do we notice such a correlation in the results so far?
- Should revisions be evaluated by both reviewers and report on consensus on what is SOLID application or not?
- Could the analysis of non RCs (as side result of our evaluation) strengthen our position?
  - changed classes would probable involve addition/deletion of code which does not reflect design changes
  - we may have enforcement of SOLID principles in new code (that is also taken into account during tagging RCs).

  – presence of refactorings would be introduced as an extra feature [refactoring-revision: yes/no]
  – requires evaluation of several revisions and we will be biased towards non-design. A better protocol could involve mixing refactoring and non-RCs before manual tagging

### B.1.2. Manual characterization of SOLID principles

The manual analysis on the introduction of SOLID principles that was carried out for our initial submission to ICSE'18 is based on the following notes:

Notice that more than one principle may apply to a RC. In any case, the application of at least one denotes a design decision. The evaluation involves checking all the changes in the RCs that the identified refactoring is part of, for instance, if the refactoring is part of a feature implementation all relevant changes are checked.

Document in generic terminology the intent of a design change with reference to SOLID principles, e.g.: An inner class absorbs responsibility from its context class...

#### B.1.2.1. *Single Responsibility Principle (SRP)*

The principle denotes that "each class, method, or module should have one reason to change". Simple heuristics on the application of SRP

- move functionality of a class to another class (Move Method) to increase its cohesion, considered in every case of functionality redistribution among classes, e.g., move of a responsibility from a subclass to its parent,
- considered also in cases of attribute relocation among classes,
- although the resulting classes usually do not have a single responsibility, any contribution towards a better situation is tagged with SRP
- replace constructor with factory method (a class acquires stricter control over the creation of its objects)
- Extracting functionality or behavior to a new method
- replacement of constants with enumerations (single responsibility for the domain of value types?)

#### B.1.2.2. *Open Closed Principle (OCP)*

The principle states that "a class should be open for extensions and closed for changes". Simple heuristics on the application of OCP:

- refactorings or changes that implement Template Method or Strategy Design patterns
- a class dependency (constructor parameter, method parameter or injected in other way) is replaced by an abstract class or interface
- a class is decoupled from a concrete implementation of a specific abstraction.
- replace conditional logic with polymorphism

#### B.1.2.3. *Liskov's Substitution Principle (LSP)*

The principle states that "supertype objects can be replaced by subtype objects without breaking the program". The principles is considered to be applied in cases of refactoring/changing members of an inheritance hierarchy:

- the public interface of a child class is narrowed down to conform to that of its parent (not found in any case in JFreeChart/Retrofit)
- method implementation changed for contract rules enforcement (hard to find, not found in any case), i.e.
  - method preconditions should not be strengthened
  - postconditions should not be weakened
  - invariants of the supertype must be maintained in the subtype
- enforcement of variance rules (parameter contravariance, return type covariance) through class parameterization (generic class)
  - e.g. a parent class that uses Object, Map, List etc. method parameters or return types is converted to parametric class and respective types replaced by class parameters.
  - the subclasses usually remove relevant type checking code.
- some cases found in Retrofit although not characterized by changes in revision metrics
- Implementation of multiple classes to implement different interfaces, such as multiple ActionListener.
- replace inheritance with composition and vice versa

B.1.2.4. *Interface Segregation Principle (ISP)*

The principle states that "clients should not be forced to depend on methods they do not use", in other words a class should implement small interfaces with targeted functionality.

The application of the principle is spotted when

- a fat interface is split into two or more simpler interfaces (found in retrofit)

B.1.2.5. *Dependency Inversion Principle (DIP)*

The principle states that "high-level modules should not depend on low level modules - both should depend on abstractions." According to R.Martin "a naïve still powerful interpretation of DIP: depend on abstractions", i.e. no variable should hold reference to concrete class, no class should derive from concrete class, no method should override an implemented method (and call the super implementation).

Simple heuristics on the application of DIP are:

- replacement of implementation inheritance to interface inheritance (remove overriding of concrete methods), although not found in JFreeChart/Retrofit
- in Retrofit, cases of refactoring/changing code to depend on abstractions is tagged as OCP
- search for "ownership inversion: clients own the abstract interfaces and servers derive from them (Hollywood Principle)"
- the implementations of an interface are distributed in a separate module, e.g. Retrofit main module owns Converter, Adapter etc. interfaces and knows nothing about implementations that are provided in separate modules