



Universidade Federal da Bahia  
Instituto de Matemática e Estatística

Programa de Pós-Graduação em Ciência da Computação

**IDENTIFYING AND ANALYZING  
SOFTWARE CONCERNS FROM  
THIRD-PARTY COMPONENTS' METADATA**

Luis Paulo da Silva Carvalho

DOCTORATE THESIS

Salvador  
2020, November, the 16th



LUIS PAULO DA SILVA CARVALHO

**IDENTIFYING AND ANALYZING SOFTWARE CONCERNS  
FROM THIRD-PARTY COMPONENTS' METADATA**

Esta Tese de Doutorado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Advisor: Manoel Gomes de Mendonça Neto

Co-advisor: Renato Lima Novais

Salvador

2020, November, the 16th

Sistema de Bibliotecas - UFBA

Carvalho, Luis Paulo da Silva.

Identifying and Analyzing Software Concerns from Third-Party Components' Metadata / Luis Paulo da Silva Carvalho – Salvador, 2020.

181p.: il.

Advisor: Prof. Dr. Manoel Gomes de Mendonça Neto.

Co-advisor: Prof. Dr. Renato Lima Novais.

Tese (Doutorado) – Universidade Federal da Bahia, Instituto de Matemática e Estatística, 2020.

1. Mining. 2. Concerns. 3. Components. 4. Static Analysis. I. Mendonça, Manoel Gomes. II. Novais, Renato Lima. III. Universidade Federal da Bahia. Instituto de Matemática e Estatística. IV. Título.

CDD – XXX.XX

CDU – XXX.XX.XXX

**LUIS PAULO DA SILVA CARVALHO**

**IDENTIFYING AND ANALYZING SOFTWARE CONCERNS FROM THIRD-  
PARTY COMPONENTS' METADATA**

Esta tese foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da UFBA.

Salvador, 16 de novembro de 2020



---

Prof. Dr. Manoel Gomes de Mendonça Neto (Orientador - PGCOMP/UFBA)



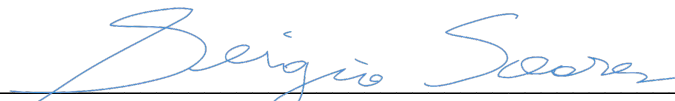
---

Profª. Drª. Laís do Nascimento Salvador (UFBA)



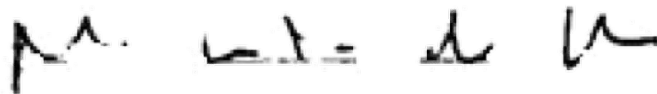
---

Prof. Dr. Cláudio Nogueira Sant'Anna (UFBA)



---

Prof. Dr. Sérgio Castelo Branco Soares (UFPE)



---

Prof. Dr. Paulo Caetano da Silva (UNIFACS)



*Para Luiz Fernando Cardeal*





## **ACKNOWLEDGEMENTS**

Agradeço a Lilian, Elaine e Edvaldo, pelo incentivo.

A João, Letícia e Gabriel pela doçura.

A Vanessa e Arthur pelo apoio.

Agradeço a meus colegas do IFBA, especialmente Djan e Crescêncio, pela cumplicidade e ao saudosíssimo, Cardeal, pela amizade (e memes) enquanto estive entre a gente.

Aos meus orientadores, Manoel e Renato, pela parceria, conselhos e paciência.

Aos colegas de PGCOMP, Ederson, Tiago Mendes, Thiago Miranda, Jorge e Felipe, pela cooperação.



*I think that it's extraordinarily important that we in computer science keep fun in computing. When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every now and then, and after a while we began to take their complaints seriously. We began to feel as if we really were responsible for the successful, error-free perfect use of these machines. I don't think we are. I think we're responsible for stretching them, setting them off in new directions and keeping fun in the house. I hope the field of computer science never loses its sense of fun. Above all, I hope we don't become missionaries. Don't feel as if you're Bible sales-men. The world has too many of those already. What you know about computing other people will learn. Don't feel as if the key to successful computing is only in your hands. What's in your hands, I think and hope, is intelligence: the ability to see the machine as more than when you were first led up to it, that you can make it more.*

—ALAN J. PERLIS



## RESUMO

Sistemas bem modularizados são mais fáceis de manter e evoluir. No entanto, não é fácil atingir uma boa modularidade. Sistemas devem ser modularizados sob diversas perspectivas e, frequentemente, existem interesses importantes que ficam espalhados ou entrelaçados entre vários módulos, os chamados interesses transversais. *Logging*, Acesso a Banco de Dados e Automação de Testes são exemplos de interesses transversais.

Os estudos sobre interesses frequentemente recorrem à identificação manual dos mesmos. Infelizmente, a identificação manual tende a ser subjetiva e imprecisa, além de exigir muito esforço. Documentos de Requisitos de Software (DRSs) e Documentos de Arquitetura de Software (DASs) podem ser usados como recursos auxiliares na análise de interesses, mas eles não são comuns. Idealmente, os desenvolvedores precisam confiar em abordagens automáticas para identificar e processar informações sobre interesses a partir de código fonte. Nesse contexto, este trabalho utiliza a injeção de componentes em projetos de software para definir um método de localização de interesses.

Em sistemas modernos, os desenvolvedores implementam módulos para lidar com as regras de negócios centrais, mas geralmente injetam componentes de terceiros na base de código para materializar interesses relacionados aos aspectos secundários do sistema. Por serem estes os tipos de interesse que mais se dispersam e se entrelaçam nos módulos dos sistemas, vimos a oportunidade de propor um método para apoiar a sua identificação e análise.

Nosso método identifica interesses a partir dos metadados que os desenvolvedores usam para injetar componentes. Em seguida, ele avalia como estes interesses se espalham e evoluem ao longo do tempo na base de código. Desenvolvemos uma ferramenta chamada *Architectural Knowledge Suite* (AKS) para automatizar o método. Usamos essa ferramenta para conduzir um estudo de pesquisa-ação com a ajuda de especialistas em desenvolvimento de software para avaliar e refinar o método. Executamos três outros estudos para caracterizar e entender como os desenvolvedores implementam interesses no mundo real.

Entre os resultados obtidos, destacamos que o método correspondeu moderadamente às expectativas dos especialistas. Notamos que é possível otimizar a captura de interesses a partir do código fonte de sistemas, se eles forem agrupados considerando similaridades entre seus contextos de uso. Percebemos que, durante a evolução dos sistemas, artefatos de código tendem a não se dedicar à implementação de interesses específicos. Identificamos oportunidades de adaptação do método para tornar possível sua aplicação sob diferentes cenários de utilização de tecnologias de desenvolvimento.

**Palavras-chave:** Mineração de Repositórios de Software, Interesses, Componentes, Análise Estática de Código.



## ABSTRACT

Well-modularized systems are easier to maintain and evolve. However, it is difficult to achieve good modularity in software systems, because developers must keep systems modular with respect to several perspectives. This involves dealing with concerns that scatter and tangle through several modules, the *crosscutting concerns*. Logging, Database Access, and Testing Automation are examples of crosscutting concerns.

Studies on concerns often resort to manual identification of interests. Unfortunately, manual identification tends to be subjective, imprecise, and effort-intensive. Software Requirement Documents (SRDs) and Software Architecture Documents (SADs) can be used as auxiliary resources to identify and analyze concerns, but they are not common assets. When they are available, there is no guarantee that they contain relevant information about concerns of particular interest. As consequence, ideally, developers should rely on automation to identify and process information about concerns over the source code. In this context, this work takes advantage of the injection of components in software projects to define a method for locating information about crosscutting concerns in software projects.

On modern systems, developers implement modules to address central business rules, but they usually inject third-party components in the codebase to materialize concerns related to secondary aspects of the system. As these are the types of concern that tend to scatter and interrelate through systems' modules, we saw an opportunity to propose a method to identify and analyzed them using injection data and metadata.

Our method first identifies concerns from the metadata that developers use to inject third-party components in their systems. Then, it evaluates how those concerns spreads, and evolve through time, over the codebase. We developed a tool named Architectural Knowledge Suite (AKS) to automate the method. We used this tool to conduct an action research study with the help of software development specialists to evaluate the reliability of our method and to refine it. We also ran three other studies using our method to process real information systems' source code, characterizing and analyzing how developers implement concerns in the real world.

Among our findings, we highlight that our method met the expectations of the specialists to a moderate degree. We perceived that grouping software projects according to their contexts of use can optimize the identification and analyses of concerns. We noticed that developers tend to mix concerns by joining references to different components through the lines of source code artifacts, but we spotted some exceptional cases. We also saw opportunities to adapt our method to expand the identification of concerns toward varied contexts of adopted software development technologies.

**Keywords:** Software Repository Mining, Concerns, Components, Static Analysis.





# CONTENTS

<b>Chapter 1—Introduction</b>	1
1.1 Objectives and Working Methodology . . . . .	4
1.2 Studies . . . . .	5
1.3 Results and Contributions . . . . .	8
1.4 Organization . . . . .	8
<b>Chapter 2—Theoretical and Technical Background</b>	9
2.1 Concerns . . . . .	9
2.2 Third-Party Components’ Metadata . . . . .	11
2.3 Third-Party Components’ Repositories . . . . .	13
2.4 Data Mining . . . . .	14
2.4.1 Software Repository Mining . . . . .	15
2.4.2 Concerns Mining . . . . .	15
2.5 Static Analysis . . . . .	16
2.6 Source Code Metrics . . . . .	16
2.7 Action Research Studies . . . . .	18
2.8 Cohen’s Kappa Agreement Coefficient . . . . .	20
2.9 Conclusion . . . . .	21
<b>Chapter 3—A Method to Extract Concerns from Third-Party Components</b>	23
3.1 Our Method . . . . .	23
3.1.1 Abstraction . . . . .	23
3.1.2 Realization . . . . .	25
3.1.3 Instantiating . . . . .	27
3.1.3.1 Identifying Java Projects’ Concerns . . . . .	29
3.1.4 Dedication to Concern . . . . .	31
3.1.4.1 Measuring Object-Oriented Projects’ Dedication to Concern	32
3.2 Applying our DtC Metric (A Worked Example) . . . . .	35
3.2.1 Measuring a High DtC . . . . .	35
3.2.2 Measuring a Moderate DtC . . . . .	37
3.2.3 Measuring a Slight DtC . . . . .	39
3.3 Discussion . . . . .	44
3.4 Related Work . . . . .	45
3.5 Conclusion . . . . .	47

<b>Chapter 4—Study I – An Action Research Study To Evaluate our Method</b>	<b>49</b>
4.1 Diagnostic . . . . .	49
4.2 Planning . . . . .	50
4.3 Actions . . . . .	50
4.3.1 Generating the Study’s Dataset . . . . .	51
4.3.2 Dataset Analysis Process . . . . .	53
4.3.3 The Raters . . . . .	55
4.4 Rounds of Action Research . . . . .	56
4.4.1 Round 1 . . . . .	57
4.4.1.1 Evaluation and Analysis . . . . .	58
4.4.1.2 Reflections and Learning . . . . .	58
4.4.2 Round 2 . . . . .	63
4.4.2.1 Evaluation and Analysis . . . . .	64
4.4.2.2 Reflections and Learning . . . . .	64
4.4.3 Round 3 . . . . .	66
4.4.3.1 Evaluation and Analysis . . . . .	66
4.4.3.2 Reflections and Learning . . . . .	67
4.4.4 A Semi-structured Interview . . . . .	74
4.5 Threats to Validity . . . . .	76
4.6 Discussion . . . . .	77
4.7 Studies that Used a Similar Approach to Validate their Methods . . . . .	80
4.8 Conclusion . . . . .	81
<b>Chapter 5—Study II – Analyzing The Evolution of the Dedication to Concern</b>	<b>83</b>
5.1 Study Definition . . . . .	83
5.2 Results . . . . .	86
5.3 Threats to Validity . . . . .	101
5.4 Discussion . . . . .	102
5.5 Conclusion . . . . .	103
<b>Chapter 6—Dissemination of Our Research</b>	<b>105</b>
6.1 Reuse of our tool – AKS . . . . .	105
6.2 Replication Packages . . . . .	108
6.2.1 Study I’s Replication Package . . . . .	108
6.2.2 Study II and III’s Replication Package . . . . .	109
6.3 Our Publications . . . . .	112
6.4 Conclusion . . . . .	113
<b>Chapter 7—Conclusion</b>	<b>115</b>
7.1 The Way We Fulfilled our Research’s Goal . . . . .	116
7.2 Future Work . . . . .	116

<b>Appendix A—Concerns Identified during Study I</b>	131
<b>Appendix B—Concerns Identified during Study II</b>	133
<b>Appendix C—Study III – Analyzing Types and Domains of Software as Transverse Dimensions</b>	137
C.1 Study Definition . . . . .	139
C.1.1 Transverse Dimension . . . . .	139
C.1.2 Agglomerations . . . . .	139
C.1.3 Code Complexity Agglomerations . . . . .	143
C.1.4 Research Questions . . . . .	145
C.2 Results . . . . .	148
C.3 Threats to Validity . . . . .	155
C.4 Discussion . . . . .	158
C.5 Related Work . . . . .	160
C.6 Conclusion . . . . .	161
<b>Appendix D—Concerns Identified during Study III</b>	163
<b>Appendix E—Study IV – Instantiating our Method under a Different Development Contexts</b>	165
E.1 Mining Concerns from NPMJS Software Projects . . . . .	166
E.2 Study Definition . . . . .	168
E.3 Results . . . . .	169
E.4 Discussion . . . . .	170
E.5 Conclusion . . . . .	173
<b>Appendix F—Concerns Identified during Study IV</b>	175
<b>Appendix G—Datasets’ Spreadsheets Format</b>	179
G.1 Study II’s Spreadsheets Format . . . . .	179
G.2 Study III’s Spreadsheets’ Format . . . . .	180
G.3 Study IV’s Spreadsheets’ Format . . . . .	181



## LIST OF FIGURES

1.1	Distribution of Concerns over 19 KLOC (BRUNTINK et al., 2004) . . . .	2
1.2	Working Methodology . . . . .	6
1.3	Studies Interconnections . . . . .	7
2.1	Our Use of the Concepts . . . . .	10
2.2	Data Mining Pipeline (AGGARWAL, 2015) . . . . .	14
2.3	Source Code to AST – Adapted from (Baojiang Cui et al., 2010) . . . . .	17
2.4	Action Research Studies Canonical Template (SUSMAN; EVERED, 1978)(DAVI- SON; MARTINSONS; KOCK, 2004) . . . . .	19
2.5	Imprecision of a Simple Percent Agreement . . . . .	20
3.1	Abstraction . . . . .	24
3.2	Realization – Adapted from (CARVALHO; NOVAIS; MENDONÇA, 2020) . . . . .	26
3.3	Method’s Instances . . . . .	28
3.4	Mining of Concerns (CARVALHO; NOVAIS; MENDONÇA, 2020) . . . . .	30
3.5	Dedication To Concern (DtC) . . . . .	31
4.1	Actions of our Action Research Study . . . . .	51
4.2	Dataset Reduction . . . . .	55
4.3	Heroic’s Tree of POM Files . . . . .	67
4.4	Comments by Rater and Round . . . . .	74
5.1	Summary of this Study . . . . .	85
5.2	DtC Agglomerations . . . . .	86
5.3	Graph Databases’ Evolution of DtC . . . . .	88
5.4	Time Series Databases’ Evolution of DtC . . . . .	95
6.1	Reuse of our tool, Architectural Knowledge Suite . . . . .	107
6.2	Reuse of our Action Research Replication Package . . . . .	110
6.3	Reuse of our Studies II and III Replication Package . . . . .	111
6.4	Reuse of our Study IV Replication Package . . . . .	112
7.1	Toward a Unified Concerns Identification Heuristic (UCIH) . . . . .	119
7.2	From DtC to Technical Debt . . . . .	120
C.1	Agglomerations of Project “P” (CARVALHO; NOVAIS; MENDONÇA, 2020) . . . . .	140
C.2	Types of Similarities . . . . .	141

C.3	Similarities between Software Types and Concerns (Density by God Class) (CARVALHO; NOVAIS; MENDONÇA, 2018) . . . . .	142
C.4	Partial Similarities between Distributed and Mobile Software Projects (Density by God Class) (CARVALHO; NOVAIS; MENDONÇA, 2018) . . . . .	143
C.5	Complexity-based Density of Agglomerations of Project “P” . . . . .	144
C.6	Summary of this Study . . . . .	148
C.7	Uniformity of Full Similarities – Transverse Dimension: Types of Software (CARVALHO; NOVAIS; MENDONÇA, 2020) . . . . .	150
C.8	Uniformity of Partial Similarities: Distributed X Service-Oriented – Transverse Dimension: Types of Software (CARVALHO; NOVAIS; MENDONÇA, 2020) . . . . .	151
C.9	Uniformity of Partial Similarities: Mobile X Service-Oriented – Transverse Dimension: Types of Software (CARVALHO; NOVAIS; MENDONÇA, 2020)	152
C.10	Uniformity of Partial Similarities: Distributed X Mobile – Transverse Dimension: Types of Software (CARVALHO; NOVAIS; MENDONÇA, 2020)	153
C.11	Uniformity of Full Similarities – Transverse Dimension: Domains of Software (CARVALHO; NOVAIS; MENDONÇA, 2020) . . . . .	154
E.1	Mining of Javascript Systems’ Concerns . . . . .	167
E.2	Chat Applications’ Concerns Cloud . . . . .	169
E.3	RPG Applications’ Concerns Cloud . . . . .	170

## LIST OF TABLES

2.1	Third-Party Components’ Repositories . . . . .	13
2.2	Kappa’s Strength of Agreement(LANDIS; KOCH, 1977) . . . . .	21
3.1	Dedication to Concern’s Metrics . . . . .	34
3.2	Non-relational Databases . . . . .	35
3.3	Measuring a Highly Dedicated Artifact . . . . .	37
3.4	Measuring a Moderately Dedicated Artifact . . . . .	40
3.5	Measuring a Slightly Dedicated Artifact . . . . .	45
4.1	Non-relational Databases (CARVALHO; NOVAIS; MENDONÇA, 2020) .	52
4.2	Format of our Agreement Dataset . . . . .	54
4.3	Evolution of our Action Research Study . . . . .	73
4.4	Tendencies regarding Raters’ Contributions . . . . .	74
4.5	Impact of Raters’ Opinions and Reactions . . . . .	78
5.1	Analyzed Projects – Adapted from (CARVALHO; NOVAIS; MENDONÇA, 2020) . . . . .	84
6.1	Our Publications . . . . .	114
C.1	Types and Domains of Software (CARVALHO; NOVAIS; MENDONÇA, 2020) . . . . .	139
C.2	Analyzed Projects (Transverse Dimension: Types of Software) (CAR- VALHO; NOVAIS; MENDONÇA, 2018) . . . . .	146
C.3	Analyzed Projects (Transverse Dimension: Domains of Software) (CAR- VALHO; NOVAIS; MENDONÇA, 2020) . . . . .	147
C.4	Concern-Based Correlations between Projects (CARVALHO; NOVAIS; MENDONÇA, 2020) . . . . .	156
C.5	Concern-Based Correlations between Projects (Continuation) (CARVALHO; NOVAIS; MENDONÇA, 2020) . . . . .	157
E.1	Analyzed Projects . . . . .	168
E.2	SSBio’s Concerns . . . . .	172
E.3	Sylus’s Concerns . . . . .	173
E.4	PopHealth’s Concerns . . . . .	174





## LISTA DE SIGLAS

<b>SAD</b>	Software Architecture Document
<b>AKS</b>	Architectural Knowledge Suite
<b>CSV</b>	Comma-Separated Values
<b>NOI</b>	Number of Imports
<b>NOIC</b>	Number of Imported Concerns
<b>NOM</b>	Number of Methods
<b>NOR</b>	Number of References
<b>ICD</b>	Imported Components' Dedication
<b>MD</b>	Methods' Dedication
<b>DtC</b>	Dedication to Concern
<b>LOC</b>	Lines of Code (metric)
<b>POM</b>	Project Object Model (metric)
<b>VCS</b>	Version Control System
<b>WMC</b>	Weighted Methods per Class (metric)



## Chapter

# 1

*Einstein repeatedly argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer – Frederick P. Brooks Jr.*

## INTRODUCTION

Software development approaches often depend on modularity as a key concept to create applications. For instance, Object-Oriented Programming (OOP) promotes ideas that favor modularity (MEYER, 1988)(MUNOZ et al., 2009). Ideally, a modular unit should encapsulate the behavior of a single concern. Concerns can be defined as anything that stakeholders consider as a conceptual unit (ROBILLARD; MURPHY, 2002)(SANT’ANNA et al., 2007). Logging of systems’ routines, Database Access, and Test Automation are examples of concerns. Specializing software modules to implement specific responsibilities is advantageous. It has the potential to guarantee that the maintenance and evolution of each concern may require modifying a single module. This can result in improvements in comparison to non-modular design (MUNOZ et al., 2009).

The tangling and scattering of concerns through source code are phenomena that affect the modularity of systems. As a consequence, they can make the developers’ work more difficult. Tangling is a state where lines of code related to different concerns are interwoven (HANNEMANN; KICZALES, 2001). Scattering means situations where the code related to a single concern spreads throughout multiple places of a system’s codebase (JUHÁR; VOKOROKOS, 2015). Crosscutting concerns are concerns that tend to scatter and tangle. As concerns crosscut, locating code fragments that implement a system’s features becomes hard because developers usually need to locate them throughout multiple code units (HE; YE, 2015). This can impact important tasks in software projects. For instance, requirements analysis that involves refining and separating concerns to understand their interrelationships (BELLOMO et al., 2014).

Keeping track of scattered and tangled concerns through development is valuable, but time and effort consuming. Depending on the size of the codebase and how impacting the scattering and tangling of concerns are, dealing with them can become counter-productive. Take Figure 1.1, for instance. It shows the distribution of concerns (Error Handling, Dynamic Execution Tracing, Function Parameter Checking and Memory Allocation Handling, in the colored lines) over 19.000 lines of code of a software component

(BRUNTINK et al., 2004). Spotting them would require developers to either remember where they placed each concern or be able to (re-)read the source code to identify them. In a worst case scenario, considering the example in the figure, developers would have to skim through many of the component's 19.000 lines of code to finish the task.



**Figure 1.1** Distribution of Concerns over 19 KLOC (BRUNTINK et al., 2004)

Hence, it is important to define methods to identify, process and analyze applications' concerns. Available methods include the use of:

1. Manual identification/mapping of concerns. It includes techniques that rely on concerns-related information manually gathered from software projects and development teams;
2. Automation based on static and dynamic analyses of software projects (DIT et al., 2013)(BERNARDI; CIMITILE; LUCCA, 2016). Static analysis refers to the use of graphs and models extracted from the source code, *e.g.*, Abstract Syntax Tree. Dynamic analysis refers to approaches that trace the execution of programs to find concerns (more details in Section 2.4.2);
3. Information stored in software documents, *i.e.*, Software Requirements Documents (SRDs) and Software Architecture Documents (SADs). In the context of requirements engineering, the concerns of central interest are requirements. In other words,

a requirement can be seen as a special kind of concern (ROSENHAINER, 2004). SADs must be clear in explaining (DÍAZ-PACE et al., 2016): (i) how functional requirements are fulfilled by responsibilities assigned to systems’ modules; and (ii) how component structures satisfy desired quality attributes. So, we consider that developers can use SRDs and SADs to identify and analyze the impact of concerns.

We believe that the aforementioned methods are relevant and can help to locate and evaluate concerns, but they may fall short from being precise and adequate under the following unfavorable circumstances (discussed in next paragraphs).

Many studies have attempted to identify concerns manually. However, even when the identification is done with the help of developers, it can lead to imprecision and discrepancies. For example, Oizumi *et al.* (2016) counted on developers to provide a list of methods and classes that implemented some software projects’ concerns. Given the large size of some of the systems, they feared that the developers would not be able to completely map the concerns. Consequently, they used a tool to perform extra identification. Later, they compared the dataset produced by the tool with another one filled by developers. Compared to the concerns identified by developers, the tool generated a longer list of data. The list was then used to increase the identification of design problems.

Developers often face the challenge of creating and updating documents because (ROBILLARD et al., 2017): (i) it is costly to write and maintain them, and (ii) considering that they are non-executable artifacts, their presence and correctness are not critical to the construction of software systems. This leads to the absence of SRDs and SADs in many software projects. Even when they are available, they may not be designed in a way to provide useful information about crosscutting concerns. There is also the possibility that they have not been updated to correctly reflect the current status of the source code.

We trust that using both static and dynamic analyses to identify concerns are good approaches. Out of the two, we favor static analysis. This comes from perceiving that source code and commits data are the two main objects of analysis when it comes to studying software repositories (FARIAS et al., 2016). Additionally, as developers store the majority of source code artifacts in Version Control Systems (VCS), static analysis of concerns can also benefit from investigating the evolution of software projects. Considering that identifying concerns through dynamic analysis requires the execution of systems, examining the evolution would require executing older versions of software projects, which can become cumbersome.

Lately, an increasing number of third-party components have become available and are being used as part of both open source and closed source developments. Third-party components (or components) can be seen as external modules which development communities make available for reuse. Developers usually implement modules to deal with core business rules, but they often inject components in the source code to materialize concerns related to secondary concerns (*e.g.*, Logging, Database Access).

In 2014 alone, Sonatype<sup>1</sup>, a repository for java components (more about repositories in Section 2.3), responded to over 17.2 billion requests for open source java-based compo-

---

<sup>1</sup><https://www.sonatype.com/>

nents from over 106.000 organizations (PALYART; MURPHY; MASRANI, 2017). Such numbers encourage us to affirm that component-based development has become a solid standard in the software industry. This comes from the fact that injecting components in the source code of systems is a way to reduce development and maintenance effort (AGÜERO; BALLEJOS, 2017)(PALYART; MURPHY; MASRANI, 2017). As a consequence, information about the nature and purpose of components have become plentiful. From an academic point of view, publications about component-oriented development increasingly attracted the attention of researchers through the 1990s and 2000s, and a lower but stable interest in the 2010s (VALE et al., 2016).

Considering that both practical and academic perspectives favor the reuse (and the study) of components, this work proposes a new way to track the occurrence of concerns. It relies on the data and metadata that developers add to software projects when they embed third-party components in software systems.

We want to support the use of this data and metadata to provide a way to extract and analyze information about concerns through the historical data of software projects. We value the identification of concerns through software projects' evolution as a way to support future studies. For instance, researchers may associate crosscutting concerns with the wealth of information residing in versioning repositories about (GİRBA; DUCASSE, 2006): reverse engineering, cost prediction, software aging, code styling and architectural decay.

## 1.1 OBJECTIVES AND WORKING METHODOLOGY

Figure 1.2 outlines our working methodology. The yellow boxes identify our objectives and the blue boxes describe how we have tackled them. Our results and findings are shown in the green boxes. Our objectives are:

1. Develop a method to identify and analyze concerns: we conceived our method to comprise a set of activities regarding the manipulation of data and metadata about third-party components (CARVALHO; NOVAIS; MENDONÇA, 2018)(CARVALHO; NOVAIS; MENDONÇA, 2020);
2. Automate our method: we instantiated our method as a tool, Architectural Knowledge Suite (AKS), to automate concerns identification activities;
3. Prepare for evaluation: after creating AKS, we used it to mine the evolution of concerns from the source code of software projects. As a result, AKS generated a comprehensive dataset which we used to conduct studies;
4. Evaluate our method: we counted on the assistance of software development specialists to evaluate our method using the dataset generated by AKS. We conducted an action research study to determine whether our method can identify and process concerns adequately;
5. Enhanced our method: we refined our method and tool to better reflect development specialists' opinions regarding the identification of concerns;

6. Evaluate the impact of concerns: we used our method and tool in studies to investigate the occurrence of crosscutting concerns in real-world software projects. We briefly describe those studies in the next section.

## 1.2 STUDIES

We ran a set of four studies during the development of this thesis. The rationale behind the studies are twofold: (i) to answer research questions about our method implementation and evolution, this is done through Studies I and II; and, (ii) to showcase the application of our method in the identification and analysis of concerns, this is done through Studies III and IV. Studies I and II are described in the main body of this thesis, as they deal with the development of our method and its metrics. Studies III and IV are described in the appendix of this thesis, as they deal with the use of our method and its metrics to mine real-world software repositories. The studies are as follows:

1. Study I – An Action Research Study to Evaluate our Method and AKS (Chapter 4): we conducted a study to evaluate our method. We based the study on a template of activities proposed by Santos and Travassos (2011) and we counted on the assistance of software development specialists to evaluate and enhance our method. In it, the specialists evaluated their perception on the relationships automatically identified by our method, among source code artifacts and crosscutting concerns;
2. Study II – The Evolution of Dedication to Concern (Chapter 5): our work introduces and evaluates a new software metric, called Dedication to Concern (DtC). DtC can be seen as the degree to which a module uses/implements a single concern. We use it to contrast the different situations in which developers embed concerns in software projects. Artifacts containing a single or few concerns may reveal development strategies that are different from others that mix different concerns. We explain and exemplify how we measure DtC in Section 3.2;
3. Study III – Types and Domains of Software as Transverse Dimensions (Appendix C): this study dates back to the initial conceptualization of our method, associating concerns with components of several types of software systems. It is a large scale software repository mining study that examines how to group software projects to extract information about concerns (CARVALHO; NOVAIS; MENDONÇA, 2018)(CARVALHO; NOVAIS; MENDONÇA, 2020);
4. Study IV – Instantiating our Method to Process Javascript Applications (Appendix E): All our studies were conducted over software systems developed using the Java language. This is a small scale repository mining study in which we demonstrate the use of our method under a different context of software development. Our intention is to show the versatility of our approach by identifying and analyzing concerns from Javascript applications.

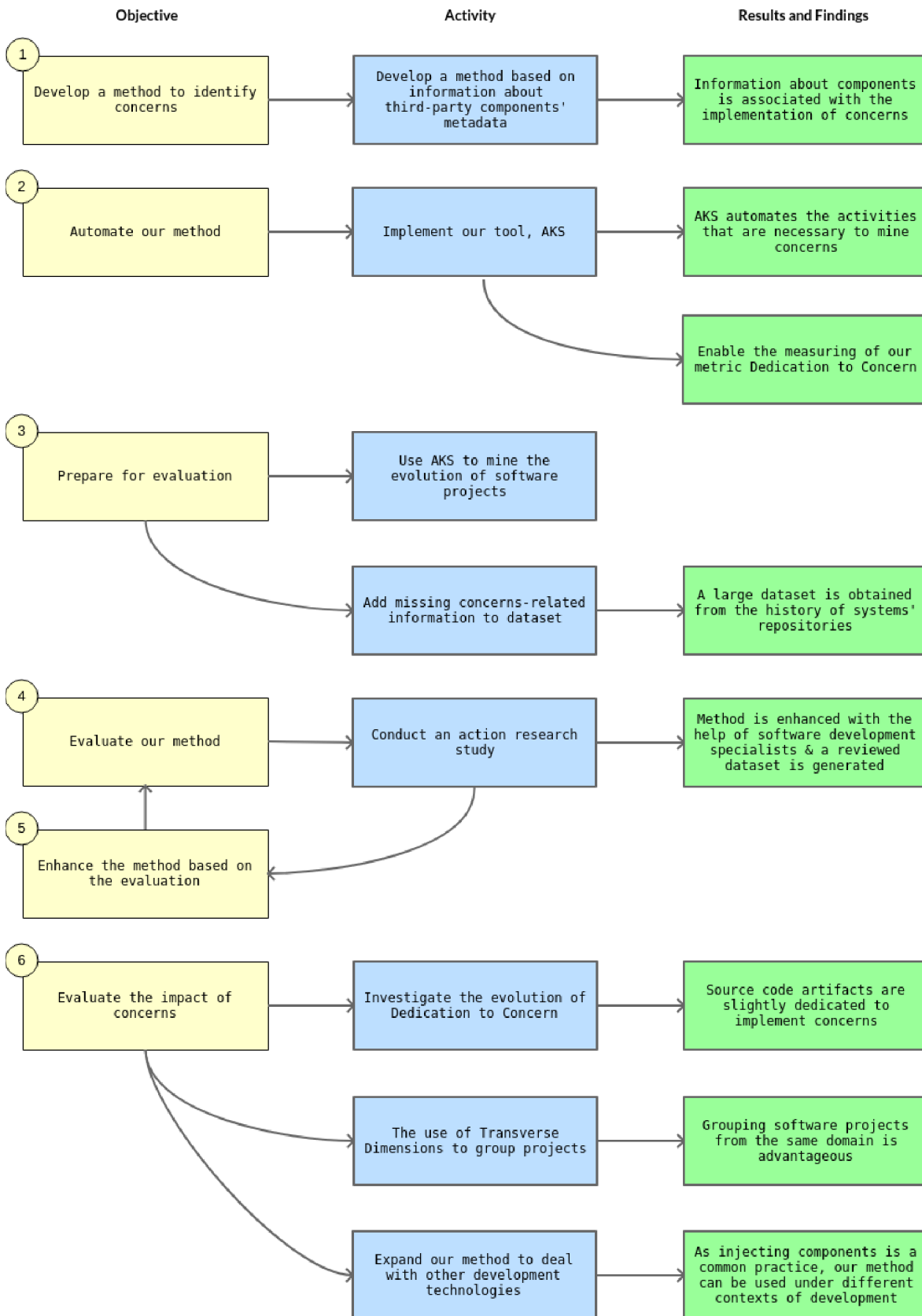
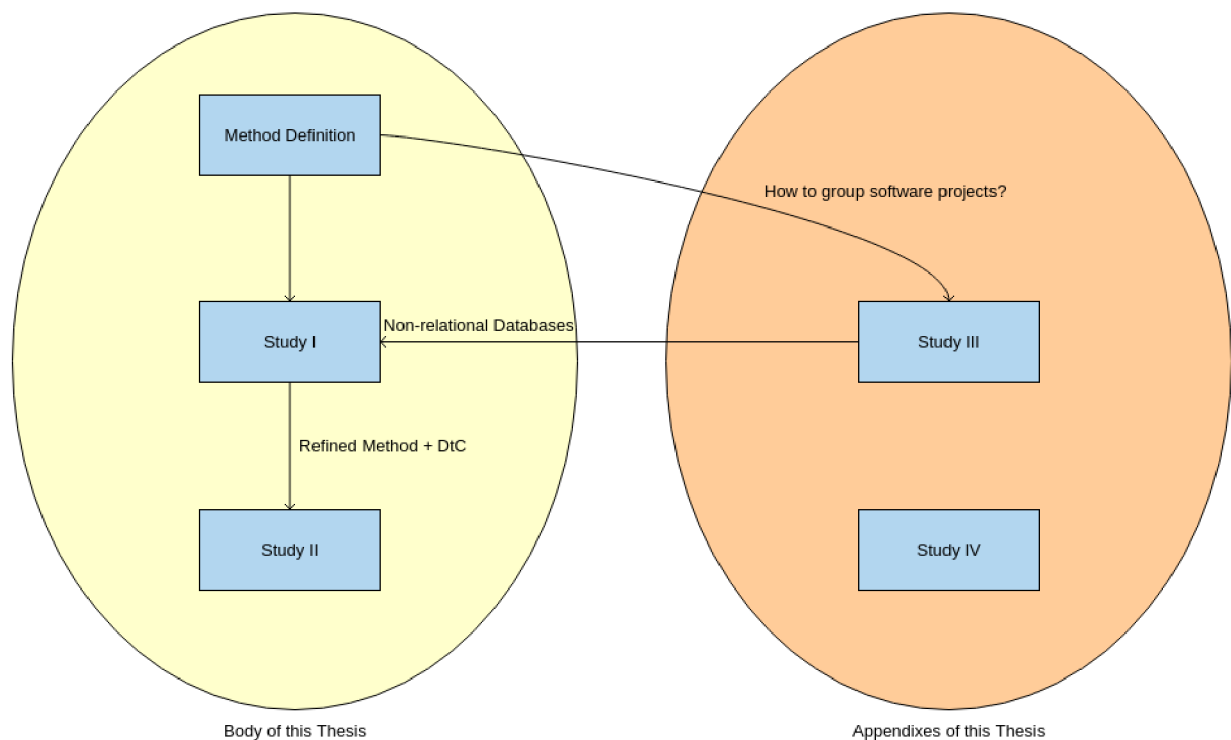


Figure 1.2 Working Methodology



It is important to point out that Study III precedes Study I and II. This means that we had not yet refined our method, with the help of software development specialists, when we conducted Study III. This implies that this thesis numbers the studies by their objectives, and not by their chronological order.

Figure 1.3 shows how the studies interconnect to each other. By defining our method's activities and automating them as much as possible, we have enabled the mining of concerns from third-party components' metadata. During Study I, we refined our method with the help of software development specialists. Study II has the purpose of examining how our metric, DtC, evolves through time. In Study III, we analyzed the association between concerns and a design problem (code complexity). Our decision to examine a specific domain of java-based systems (non-relational databases) in studies I and II comes from Study III. Study IV shows that our method is generic enough to embrace a different context of development: javascript projects.



**Figure 1.3** Studies Interconnections

One important remark about our studies is that we rely heavily on visualizations to present our findings. Visualizations are useful to explore and gain insights over large amounts of information (MAGNAVITA; NOVAIS; MENDONÇA, 2015)(FRANCESE; RISI; SCANNIELLO, 2015)(MENDES et al., 2015)(NOVAIS; SANTOS; MENDONÇA, 2017). By extracting evolutionary data, AKS produces an extensive quantity of data during its crosscutting concerns analyses. We use visualization to explore the information that AKS mines from the software projects analyzed throughout this work. We apply visualizations to present our results in a concise and intuitive way.

### 1.3 RESULTS AND CONTRIBUTIONS

The main contribution of this thesis is the development of a scalable method to identify and analyze crosscutting concerns in software development projects. Our studies have shown that our method can support the identification and analysis of concerns by taking advantage of third-party components' metadata. Our classification of concerns and measurement of DtC were considered acceptable, as perceived by the specialists that participated in our action research study. The following are other contributions of this thesis:

1. We developed a tool, AKS, to automate our method. The tool is readily available for reuse. AKS can help researchers to perform further studies. We also believe that it can already be used by practitioners to run useful analysis of crosscutting concerns, during software development activities;
2. We produced a comprehensive dataset which comprises: (i) a detailed classification of concerns which we extracted from software projects; and (ii) a categorization of concerns regarding their relationship with systems' source code artifacts (Dedication to Concern). Researchers may reuse our dataset to replicate our studies and/or conduct investigations other than the ones we address in this thesis.

We emphasize other potential contributions of our method and this thesis's studies findings. We believe that our research can support developers to carry important software-related activities, such as: (i) understanding the causes and effects of crosscutting concerns as their presence can hinder the modularization of systems (KICZALES, 1996)(HE; YE, 2015); (ii) refactoring of systems regarding the occurrence of concerns, which has been deemed as an important task in software maintenance (SILVA et al., 2009)(NUÑEZ-VARELA et al., 2017a); and (iii) addressing software architecture issues that are associated with the definition of concerns, *e.g.*, mapping the currently implemented concerns to the reference architecture to verify conformance while avoiding architecture erosion (ADAMS; JIANG; HASSAN, 2010).

### 1.4 ORGANIZATION

This remainder of this text is organized as follows. Chapter 2 discusses the theoretical and technical foundations of our research. Chapter 3 describes our method. Chapters 4 and 5 present the results of the studies that we conducted to evaluate the applicability of our method and to answer some of its research questions. Chapter 6 explains our strategies to disseminate our method, tool, datasets, and the results of our studies. Our final remarks can be found in Chapter 7. Appendixes C and E describe, respectively, Studies III and IV, showing the identification and analysis of crosscutting concerns in the large, in real world projects.

*The computing scientist's main challenge is not to get confused by the complexities of his own making –  
Edsger Dijkstra*

## THEORETICAL AND TECHNICAL BACKGROUND

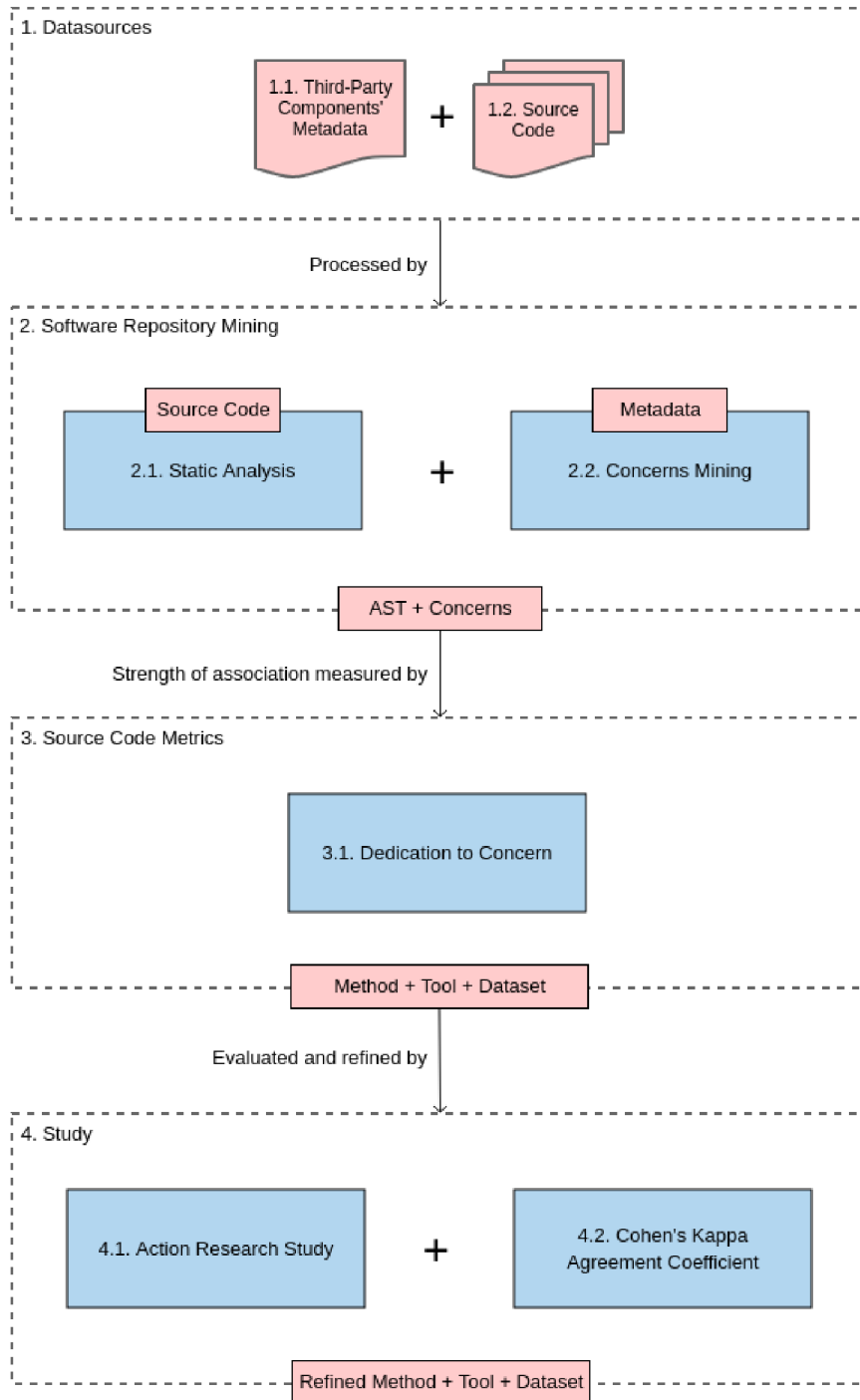
This chapter introduces the main concepts associated with our research. Figure 2.1 depicts the use of those concepts in our work and provides a visualization of what we discuss in the next sections.

Our work starts by extracting **concerns** from **third-party components' metadata** and associating them with **source code** artifacts – those topics are discussed in Section 2.1 and Section 2.2. To achieve this, our work apply techniques from a specific branch of data mining, called **software repository mining** – this topic is discussed in Section 2.4.1. In this theme, we combine **static analysis** of source code and **concerns mining** – those topics are discussed in Section 2.5 and in Section 2.4.2, respectively. This results in a mapping between **Abstract Syntax Trees** and **concerns** – this topic is explained in Section 2.5 as well. Usually, we combine this mapping with **software metrics** extracted from the static analysis – this is discussed in Section 2.6. As we are introducing a new way to recover information about concerns, we also needed to evaluate and refine our method, tool, and dataset. For this, we conducted an **Action Research Study** – this topic is explained in Section 2.7. Through the phases of our study, we applied the **Cohen's Kappa Agreement Coefficient** to assure the alignment among the specialists' opinions – this topic is explained in Section 2.8.

### 2.1 CONCERNS

Concerns can be defined as anything that stakeholders consider as a conceptual unit in a software system (ROBILLARD; MURPHY, 2002)(SANT'ANNA et al., 2007). In other words, concerns refer to each constituent part of a software system that comprises a set of interrelated functionalities. Adding pieces of code to automate logging, database access, security, encryption, user interface and data streaming are examples of how developers implement concerns.

The creation of software systems can be seen as a constant addition and evolution of concerns that developers must implement in response to stakeholders' needs (JUHÁR;



**Figure 2.1** Our Use of the Concepts

VOKOROKOS, 2015). During maintenance and re-engineering cycles, developers may need to locate concerns in the source code to perform important tasks. For instance, they may use this information to propagate bug fixes to the whole implementation of a concern (EADDY et al., 2008). If done manually, the location of concerns can be tedious and

subjective. Preferably, developers should apply mining techniques to find and analyze concerns with precision and efficiency (ADAMS; JIANG; HASSAN, 2010).

One important applicability of concerns is related to their use in splitting the complexity of a system into smaller and more manageable modules. If well done, separation of concerns can impact the overall quality of the system in a positive way by reducing the complexity, increasing the intelligibility, and facilitating reuse, evolution and customization (HE; YE, 2015). Separation of concerns can be seen as something that developers should pursue to guarantee modularity. However, this is often neglected. For instance, many object-oriented systems suffer from the scattering and tangling of concerns (HANNEMANN; KICZALES, 2001)(NUÑEZ-VARELA et al., 2017a). Some concerns cannot be neatly separated in objects and, hence, they spread (or scatter) across several modules (JUHÁR; VOKOROKOS, 2015). Tangling is related to situations in which lines related to different concerns are interwoven through source code (HANNEMANN; KICZALES, 2001). Crosscutting concerns are concerns that tend to scatter and tangle. “Logging”, for example, affects every logged part of a system. Thus, “Logging” can crosscut through many software modules (BERNARDI; CIMITILE; LUCCA, 2016). Maintaining a crosscutting concern means modifying each fragment of the scattered/tangled code realizing that concern. Eventually, this may increase the coding time, error proneness, and the maintenance cost (MUNOZ et al., 2009). Developers must be aware of and manage crosscutting concerns with attention.

## 2.2 THIRD-PARTY COMPONENTS' METADATA

According to Merriam-Webster dictionary, “metadata” is *data that provides information about other data*<sup>1</sup>. Developers often depend on metadata to store important information about varied aspects of their software projects (*e.g.*, classpaths, integration with other projects) or to automate the execution of important tasks (*e.g.*, tests, deployment). Among such metadata, we have focused on the ones that developers use to inform which components must be embedded in their systems. Common advantages achieved by reusing components are related to the addition of previously implemented and previously tested functionalities. Consequently, they can reduce effort and improve quality during software development (SHATNAWI et al., 2017).

We are interested in Project Object Model (POM) and Gradle files that developers use to inject components in java-based software projects (AGÜERO; BALLEJOS, 2017)(PALYART; MURPHY; MASRANI, 2017). This comes the fact that such files have become plentiful because many java developers have adopted them to encapsulate information about components used in their projects. Listing 2.1 contains some lines extracted from a POM file<sup>2</sup>:

**Listing 2.1** Example of a POM file

```
1 <project>
2   ...
3   <properties>
```

<sup>1</sup><https://www.merriam-webster.com/dictionary/metadata>

<sup>2</sup>Adapted from <https://maven.apache.org/guides/introduction/introduction-to-the-pom.html>

```

4     <mavenVersion>2.1</mavenVersion>
5 </properties>
6 <dependencies>
7   <dependency>
8     <groupId>org.dbunit</groupId>
9     <artifactId>dbunit</artifactId>
10    <version>2.6.0</version>
11  </dependency>
12  <dependency>
13    <groupId>org.springframework</groupId>
14    <artifactId>spring-test</artifactId>
15    <version>5.1.3.RELEASE</version>
16  </dependency>
17 </dependencies>
18 ...
19 </project>

```

Components stored in POM and Gradle files are uniquely identified by their (HERNÁNDEZ; COSTA, 2015): (i) *group id*: company, team, organization; (ii) *artifact id*: unique id, one for each component; and (iii) *version*: version of the component. For instance, in the POM file shown in Listing 2.1, a component used for test automation, **dbunit**, is identified by its *group id*, “org.dbunit” (line 8), *artifact id*, “dbunit” (line 9), and *version*, “2.6.0” (line 10). The file also encapsulates information about a second test component, **spring-test**, which has a different *group id*, “org.springframework” (line 13), *artifact id*, “spring-test” (line 14), and *version*, “5.1.3.RELEASE” (line 15). Modern development environments (*e.g.*, Eclipse Platform<sup>3</sup>) are capable of parsing the POM to download and insert both components as dependencies in software projects.

Alternatively, developers may store metadata about components in Gradle files. Listing 2.2 exemplifies how developers can add the same aforementioned components (**dbunit** and **spring-test**) to this type of file (lines 8 and 9). It is important to notice that both POM and Gradle require the unambiguous identification of components by inserting their respective ids.

**Listing 2.2** Example of a Gradle file

```

1 plugins {
2   id 'java-library'
3 }
4 repositories {
5   jcenter()
6 }
7 dependencies {
8   testImplementation 'org.dbunit:dbunit:2.6.0'
9   testImplementation 'org.springframework:spring-test:5.1.3.
    RELEASE'
10 }

```

We emphasize that files containing third-party components’ metadata are not restricted to java-based systems. As the injection of components has become a widespread

---

<sup>3</sup><https://www.eclipse.org/>

practice, it is possible to find them in many other types of software projects. In Appendix E we cover examples of similar metadata files that we found in systems written in javascript, PHP, python and ruby. In Chapter 3, we explain how the unambiguity of components' ids enabled us to precisely associate concerns with components.

## 2.3 THIRD-PARTY COMPONENTS' REPOSITORIES

We can fit the origins of third-party components into four categories (BADAMPUDI; WOHLIN; PETERSEN, 2016): (i) in-house developed components: components are developed within the companies, (ii) Components Off-The-Shelf (COTS): commercial components that are bought from external vendors, (iii) Open Source Software (OSS): components obtained from OSS communities, and (iv) outsourced components: the development of components is outsourced (or subcontracted). Regarding OSS, we focus on the MAVEN community<sup>4</sup> in this thesis. MAVEN is very popular in supporting developers who want to reuse java-based components (PALYART; MURPHY; MASRANI, 2017).

The MAVEN community created an on-line repository to provide information about components: the MVNRepository<sup>5</sup>. In other words, MVNRepository is a web portal responsible for indexing useful information about java-based OSS-oriented components. It has attracted the attention of developers because of the popularity of the languages supported by the Java Virtual Machine (JVM), which includes not only java, but scala, kotlin and clojure (VELÁZQUEZ-RODRÍGUEZ; ROOVER, 2020).

The combination of programming languages and the reuse of components has spawned several other communities and repositories. This means: creating and maintaining on-line information about components has become a standard because the reuse of components is a common procedure regardless the programming language. In Table 2.1 we highlight other repositories and their respective languages.

**Table 2.1** Third-Party Components' Repositories

Programming Language	Repository	Details and Examples of Use in...
Javascript	<a href="https://www.npmjs.com/">https://www.npmjs.com/</a>	Appendix E
Python	<a href="https://pypi.org/">https://pypi.org/</a>	Section E.4
PHP	<a href="https://packagist.org/">https://packagist.org/</a>	Section E.4
Ruby	<a href="https://rubygems.org/">https://rubygems.org/</a>	Section E.4
R	<a href="https://www.rdocumentation.org">https://www.rdocumentation.org</a>	-
Dart	<a href="https://pub.dev/">https://pub.dev/</a>	-
C++	<a href="https://conan.io/center/">https://conan.io/center/</a>	-
Lua	<a href="https://luarocks.org/">https://luarocks.org/</a>	-
Rust	<a href="https://crates.io/">https://crates.io/</a>	-

<sup>4</sup><https://maven.apache.org/>

<sup>5</sup><https://mvnrepository.com/>

By consulting MVNRepository developers can get access to a precise set of information about third-party components: their categories, descriptions, tags/keywords, list of versions/releases, known bugs, limitations, developers, etc. In Chapter 3, we describe how we associate the metadata about components (as described in Section 2.2) with some extra information collected from a repository (MVNRepository) to create our method.

## 2.4 DATA MINING

According to Hand and Adams (2014), Data mining can be defined as the “science of extracting useful information from large datasets or databases”. It comprises studies and methods related to collecting, cleaning, processing, analyzing and gaining insights from data (AGGARWAL, 2015). This is usually achieved via a pipeline of processing, where raw data is collected, cleaned and transformed into a standardized format. The pipeline is conceptually similar to that of an actual mining process from a mineral ore to the refined end product. The term “mining” derives its roots from this analogy (AGGARWAL, 2015). Figure 2.2 illustrates how the data mining pipeline works.

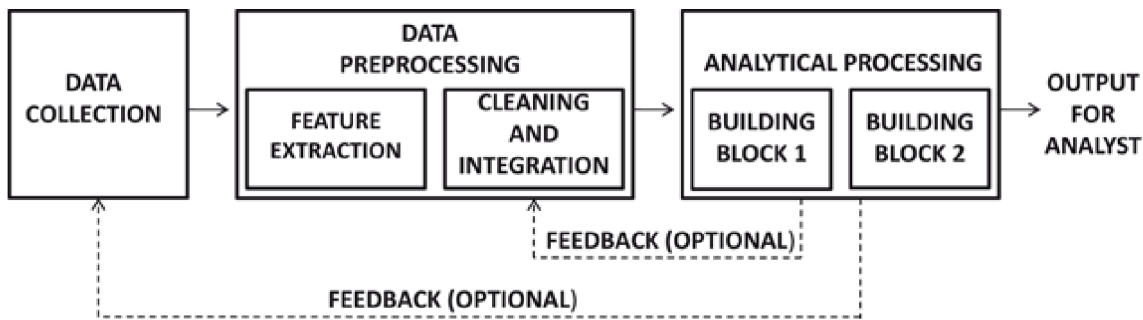


Figure 2.2 Data Mining Pipeline (AGGARWAL, 2015)

**Data collection** is the process of finding reliable datasources and capturing data from them. When data are collected, they may not be in a form that is suitable for analysis, so **Data Preprocessing** comprises activities (**Feature Extraction** and **Cleaning and Integration**) that reshape the gathered information into insightful datasets. The last activity, **Analytical Processing**, concerns itself with designing effective analytical methods from the processed data, *i.e.*, the data mining specialist must decide about the best strategies to concisely and coherently arrange, measure and present the data.

One noticeable aspect of the pipeline is that it is a cyclical mechanism. While data mining specialists collect and process the data, they can refine their work with the help of feedback from researchers and practitioners. This is crucial for our research. For instance, as we mined data to support our action research study (described in Section 4), we benefited from software development specialists’ comments and recommendations to enhance our concerns identification method and dataset through rounds of evaluation.



### 2.4.1 Software Repository Mining

In the context of our work, we deal with a specific type of data mining: the mining of software repositories (MSR). MSR focuses on uncovering interesting and useful information about software projects while extracting and analyzing available data from software repositories (HASSAN, 2008). Repositories, such as Github<sup>6</sup> and Gitlab<sup>7</sup>, contain large amount of software historical data that can include valuable information about information systems' source code, defects and issues (FARIAS et al., 2016). As third-party components are vital part of software projects, metadata about them is also frequently added to the historical data of software repositories.

Mainly, this thesis focuses on the mining of one specific information from software projects' repositories: concerns. This is achieved by associating third-party components' metadata with other useful information extracted from system's source code artifacts, for example, software metrics like Number of Lines of Code (more details in Section 2.6) that are added, changed or deleted during the evolution of software projects (CANFORA; CERULO, 2005).

### 2.4.2 Concerns Mining

Software mining techniques have explored both static and dynamic analysis of software projects to extract useful information on concerns (DIT et al., 2013)(BERNARDI; CIMITILE; LUCCA, 2016). Static analysis comprises the use of parsing and lexical analysis to extract information, graphs, and models, such as *Abstract Syntax Trees*, directly from the source code. Dynamic analysis uses approaches that trace the execution of the software programs to gather information its workings. The following are examples of static and dynamic analysis applications that can support concern mining (DIT et al., 2013)(MARÇAL et al., 2016):

1. Clone detection techniques: clones are generally defined as a code fragment identical or similar to another code fragment. This technique uses the identification of code cloning to spot concerns;
2. Fan-in analysis: the fan-in value is computed as the number of calls to a method. Methods with high fan-in value have considerable chances of being classified as concerns;
3. Graph-based techniques: graphs allow the representation of elements of code as a set of nodes and edges. The edges link nodes to each other. The links can be considered as relationships between nodes. An approach can extract information about concerns from the relationships;
4. Clustering analysis: clustering analysis groups data with similar characteristics. The resulting clusters can be analyzed to determine which ones are part of concerns;

---

<sup>6</sup><https://github.com/>

<sup>7</sup><https://about.gitlab.com/>

5. Textual Feature Location: approaches based on the use of textual search and Natural Language Processing (NLP) link textual descriptions of features given by developers to parts of the source code where the features are implemented;
6. Techniques based on development history: software version control systems provide a rich collection of historical data related to changes in software artifacts.

It is important to point out that the mentioned methods are not mutually exclusive. Actually, the combination of static and dynamic approaches is a powerful strategy in many situations (DIT et al., 2013). In this thesis, however, we focus on static analysis of code. Section 2.5, below, discusses it in greater detail. Chapter 3 introduces our concerns identification and analysis method, which applies **static analysis** of source code artifacts retrieved from the **development history of software projects**.

## 2.5 STATIC ANALYSIS

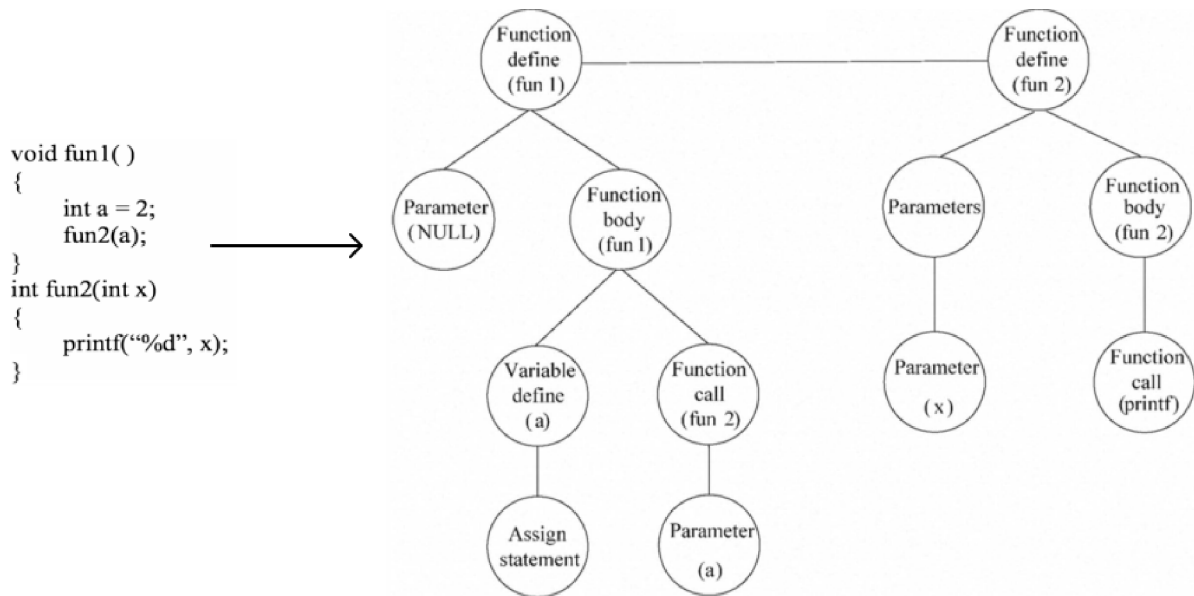
Static analysis can be defined as any process of assessing source code without executing it (DELEV; GJORGJEVIKJ, 2017). The information reported from static analysis can be used to improve code quality, security, robustness, and to mitigate problems. It is often applied in finding flaws in programs logic that can be potential causes of bugs. Developers can also use it to learn about hidden features of a programming language that may lead to unusual behavior of systems (EMANUELSSON; NILSSON, 2008)(DELEV; GJORGJEVIKJ, 2017).

One way to automate static analysis is to process programs' source code as Abstract Syntax Trees (AST). Rather than resorting to text-oriented processing, AST enable storing information about source code elements in a high level structure (Baojiang Cui et al., 2010). Figure 2.3 illustrates the representation of a piece of code (on left side of the figure) as an AST (on the right side). Software tools can statically parse a program's source code to produce an AST. When a specific sequence of tokens match a particular rule (*e.g.*, function definition, parameters and variables declarations), they generate a syntax tree node (the circles in the figure) and record the corresponding node type, as well as its position in the source code.

We rely on Repository Miner (RM) (MENDES et al., 2017) to extract AST from software projects. RM has been mentioned and used in studies that seek to perform static analysis and to identify and investigate phenomena related to source code comprehension, design problems and technical debt (MENDES et al., 2015)(IBIAPINA et al., 2018)(GOMES et al., 2019)(DIAS et al., 2019)(KHOMYAKOV et al., 2019)(MENDES et al., 2019)(FARIAS et al., 2020).

## 2.6 SOURCE CODE METRICS

Nunez *et. al.* (2017b) consider that source code metrics (or software metrics) are central to aid software measurement processes. Software measurement is a task often carried out when it is necessary to assess the quality of systems. In general, metrics support a wide range of activities concerned with producing numbers and conceptualizations



**Figure 2.3** Source Code to AST – Adapted from (Baojiang Cui et al., 2010)

that characterise properties of software code. The properties include quantitative and qualitative aspects of software development and quality control (FENTON; NEIL, 2000).

According to Fentom and Neil (2000), it is possible to summarize the motivation behind most software metrics into two main categories: (i) the desire to assess or predict effort/cost of development processes, and (ii) the desire to assess or predict quality of software. The key in both cases has been the assumption that “software size” should drive predictive models. The first metric used to do this was the Lines of Code (LOC) metric. We exemplify LOC in Listing 2.3 by measuring it from the same source code excerpt exhibited in Figure 2.3. Supposing the two functions, *fun1* and *fun2*, reside in the same artifact “A”, the total LOC of “A” is 9.

**Listing 2.3** Measuring Lines of Code (LOC)

```

1 void fun1()
2 {
3     int a = 2;
4     fun2(a);
5 }
6 int fun2()
7 {
8     printf("%d", x);
9 }
```

Other metrics can be obtained from systems’ AST. For instance, the Number of Local Variables (LVAR) (NUÑEZ-VARELA et al., 2017b) is a metric that can be calculated with the help of the AST. Regarding the AST in Figure 2.3, the LVAR of *fun1* is 1 because its syntax tree has only one node that points to a variable declaration (“Variable define (a)”). The LVAR of *fun2* is 0 because its tree does not contain a reference to the declaration of a variable.

## 2.7 ACTION RESEARCH STUDIES

Action research studies stem from the principle of cyclical field intervention, which allows the testing and refinement of theories in practice (BASKERVILLE, 1999)(PUHAKAINEN; SIPONEN, 2010). It is a clinical method, aimed at creating change and solving practical problems through research (BASKERVILLE; MYERS, 2004). Action Research has its origins associated with the early interventionist practices in the course of social–technical experiments. The initial stimulus for the rise and design of action research objectives came from a generalized difficulty in translating the results of social research into practical actions (SUSMAN; EVERED, 1978)(SANTOS; TRAVASSOS, 2011).

Regarding the use of action research in software engineering, Santos and Travassos (2011) ran a survey that covered 5 years of publications, from 2005 to 2010, and noticed a smooth increase of studies that reported the application of action research methods. They see it as an alternative to run investigations that depend on direct access to the know-how of practitioners, which is not often achieved by surveys and controlled studies. The survey also revealed the two main domains of use of action research in software engineering: (i) a more socially-oriented application (*e.g.*, Management and Software Engineering Processes) and (ii) a technical applicability (*e.g.*, Software Construction and Programming Environments).

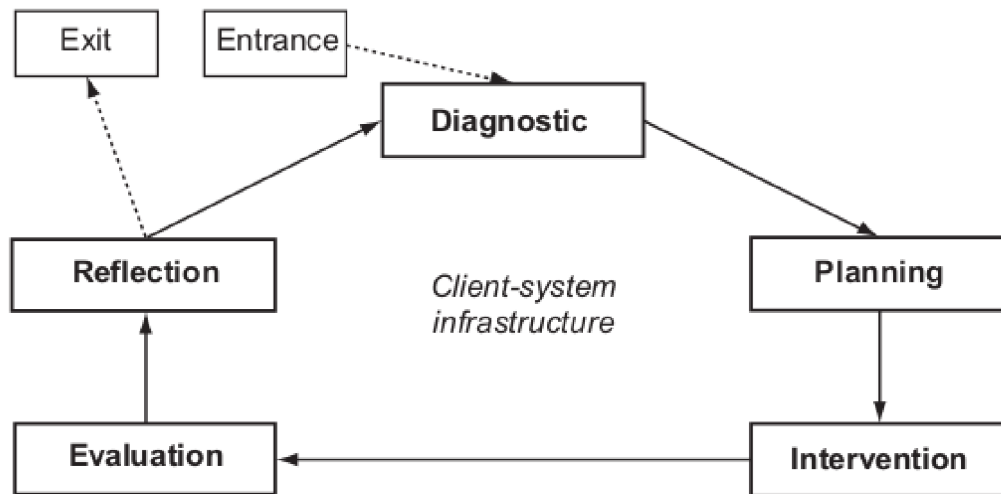
Action research methods focus on: (i) improvement of the practice, (ii) on learning and (iii) emphasis on what practitioners do. Compared to other research methodologies, action research places more attention on intervention and learning (STARON, 2020). This becomes evident in the template of action research activities proposed by Susman and Evered (1978) (in Figure 2.4). The template has been considered canonical<sup>8</sup> and reflects the dual intention of action research: learning and adapting through observation and intervention.

One first noticeable aspect is the cyclical execution of its activities/stages, *i.e.*, from its starting point (the **Diagnostic** stage) the study can be restarted as many times as it is necessary to fulfill its goals. The following are descriptions of the template’s activities (SANTOS; TRAVASSOS, 2011)(STARON, 2020):

1. Diagnostic: consists of exploring the research field, stakeholders, and their expectations. In this stage, researchers define the research’s theme which is represented by the designation of the practical problem and knowledge area to be addressed;
2. Planning: stage where actions are defined to the faced circumstances. The definitions are guided by hypotheses portraying the researchers’ formulated assumptions about possible solutions and results;
3. Intervention: corresponds to the planned actions implementation. Examining, discussing, and making decisions about the investigation process are central elements of the intervention phase;

---

<sup>8</sup>Merriam-Webster dictionary defines “canonical” as: “conforming to a general rule or acceptable procedure”



**Figure 2.4** Action Research Studies Canonical Template (SUSMAN; EVERED, 1978)(DAVISON; MARTINSONS; KOCK, 2004)

4. Evaluation: stage where the intervention effects are analyzed considering the theoretical background used as basis to the actions definition;
5. Reflection: involves the dissemination of acquired knowledge among participants and other organization departments. The learning experience is facilitated by the previous collaboration among participants and researchers in the technical topics.

Baskerville (1999) exemplifies an action research that had the intention to complete a system's analysis. Early development efforts failed to finish the task because of a complicated database design and requirements. These failures complicated further analysis as the users had grown hostile and suspicious of analysts and designers. An action research team was formed to solve the problem. From the **diagnostic** perspective, they found out that the project was defeated mainly by (i) a set of large data classes and volume of data, and (ii) the high volatility in the organizational environment. Then, in the **planning** phase, they decided to adopt prototyping as a method to conduct the system's analysis. **Intervention** took place in the form of rapid, brief interviews with the users, and fast prototyping cycles of the database design. With each intervention users assessed the prototypes and researchers took advantage of their feedback, *i.e.* from user's **evaluation** of the prototypes researchers enhanced the system's analysis based on their opinions. After the last evaluation, researchers ended up with the following **reflection**: the use of prototypes proved significant to solving the immediate problem setting and the analysis was completed.

In Chapter 4, we report an action research study that we conducted with the purpose of evaluating and refining our method.

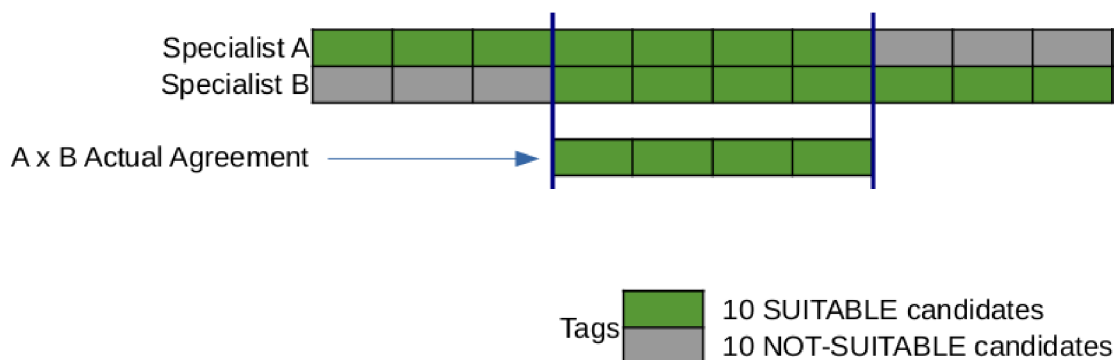
## 2.8 COHEN'S KAPPA AGREEMENT COEFFICIENT

The Cohen's Kappa Coefficient is useful when it is necessary to use descriptive statistics to summarize the agreement between two assigners (or judges, or raters) across a number of objects (*e.g.*, things, statements, calculations) (COHEN, 1960)(BRENNAN; PREDIGER, 1981). Kappa can calculate the proportion of the agreement while correcting it for chance (COHEN, 1960). We further illustrate a fictitious example to show how kappa works and to explain its concepts.

Suppose it is necessary to determine, out of a list of 100 people, which ones are electable for financing a new car. Two finance specialists (specialist "SPEC-A" and "SPEC-B") are hired to perform the selection by tagging the list after examining a dataset about the candidates. At the end of the selection, each person must be tagged as either SUITABLE or NOT-SUITABLE. In addition, to increase the confidence on the results, the specialists are not allowed to pair-tag the dataset. In other words, "SPEC-A" and "SPEC-B" must tag the same dataset independently, while avoiding consulting each other.

After tagging the dataset, the specialists must return it to finish the selection process. Considering that "SPEC-A" approved (*i.e.*, tagged as SUITABLE) 70 people and "SPEC-B" also found 70 SUITABLE people, a simple percent agreement calculation would state that the specialists tagged 70% of the candidates as SUITABLE. However, this type of calculation can lead to imprecise results, which can cause the approval of financing plans for non-suitable people. Figure 2.5 depicts this situation.

Assuming that "SPEC-A" approved the first 70 candidates of the dataset and "SPEC-B" the last 70 ones, the actual agreement would comprise the intersection of these two groups, with only 40 candidates (or 40% of them) being considered SUITABLE by "SPEC-A" and "SPEC-B" simultaneously.



**Figure 2.5** Imprecision of a Simple Percent Agreement

Kappa corrects the imprecision of agreements (as the one exhibited in Figure 2.5) by considering all possible combinations of tags provided by the two raters: (i) all candidates who "SPEC-A" tagged as SUITABLE and "SPEC-B" as NOT-SUITABLE; (ii) all candidates who "SPEC-A" tagged as NOT-SUITABLE and "SPEC-B" as SUITABLE; (iii) all candidates who "SPEC-A" and "SPEC-B" tagged as SUITABLE; and (iv) all

candidates who “SPEC-A” and “SPEC-B” tagged as NOT-SUITABLE.

The R-language<sup>9</sup> script shown in Listing 2.4 can be used to replicate this illustrative example. The script is capable of loading the set of agreements and disagreements shown in Listing 2.5 (lines from 1 to 4) before processing it through kappa (line 7).

**Listing 2.4** A R-language Script to Calculate Kappa

```

1 rater1_data <- replicate(70, 1)
2 rater1_data <- append(rater1_data, replicate(30, 0))
3 rater2_data <- replicate(30, 0)
4 rater2_data <- append(rater2_data, replicate(70, 1))
5
6 agreement <- cbind(rater1_data, rater2_data)
7 kappa2(agreement)

```

Executing the script outputs the lines below (in Listing 2.5). Acquiring a negative value as result (line 3) means that “SPEC-A” and “SPEC-B” disagreed about the selection of candidates.

**Listing 2.5** Kappa’s Agreement Coefficient

```

1 Subjects = 100
2 Raters = 2
3 Kappa = -0.429

```

A perfect result would be any value near to 1.0 (in a 0.0 to 1.0 range), as described in Table 2.2.

**Table 2.2** Kappa’s Strength of Agreement(LANDIS; KOCH, 1977)

Kappa	Strength of Agreement
<0.00	Poor
0.00-0.20	Slight
0.21-0.40	Fair
0.41-0.60	Moderate
0.61-0.80	Substantial
0.81-1.00	Almost Perfect

Our work uses the Cohen’s Kappa Coefficient to check the agreement among the experts and our method on the identification of concern, and on software artifacts’ dedication to concerns, in our action research study.

## 2.9 CONCLUSION

Our work focus on the identification and analysis of **concerns on third-party components’ metadata**. We fulfill our goal by **mining software repositories** and extracting **software metrics** from **abstract syntax trees**. We evaluate our method through an

<sup>9</sup><https://www.r-project.org/>

**action research study** that uses the **kappa coefficient** to quantify the agreement among the method and software professionals.

This chapter introduced those concepts, as the the main theoretical and technical aspects that we refer to throughout the remainder of this dissertation. We must emphasize that the list of concepts presented here is not exhaustive and we mention some others in the next sections.

Next Chapter introduces and describes our method of extracting concerns from third party components.



*Computer Science is a science of abstraction: creating the right model for a problem and devising the appropriate mechanizable techniques to solve it – Alfred Aho*

## A METHOD TO EXTRACT CONCERNS FROM THIRD-PARTY COMPONENTS

Our work seeks to mine a large quantity of data to assertively base our findings, conclusions, and discussions regarding software concerns. For this end, we developed a method to semi-automatically identify and analyze concerns in the large, using third party components injection in modern software systems (CARVALHO; NOVAIS; MENDONÇA, 2018)(CARVALHO; NOVAIS; MENDONÇA, 2020).

Our method takes advantage of concerns-related metadata found in Open Source Systems (OSS). We favor the analysis of OSS because they have been consistently used to (FARIAS et al., 2016): (i) examine important characteristics related to development processes, (ii) extract metrics, and (iii) assess the quality of software artifacts. Additionally, many open source systems have large repositories of historical data, comprising several changes, releases, and information about third-party components.

Next sections explain our method and exemplify its use.

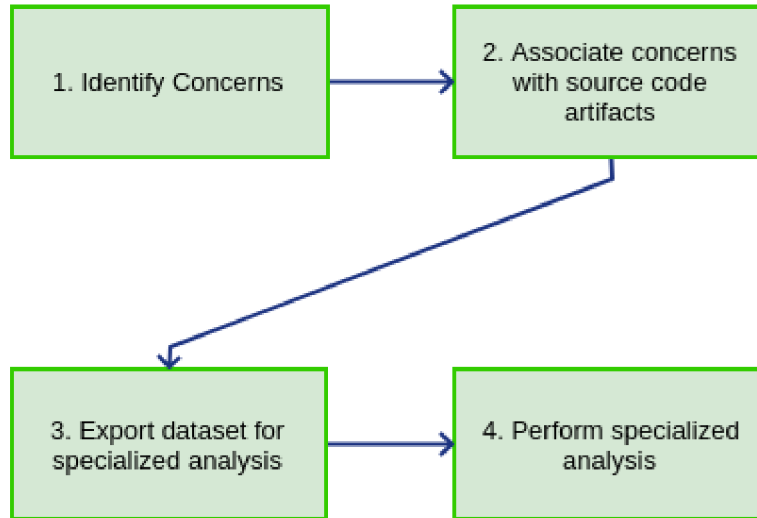
### 3.1 OUR METHOD

Our method comprises of a set of activities that we designed to take advantage of third-party components' metadata. We have split the activities into abstract and semi-automated ones. We present them through the next sections.

#### 3.1.1 Abstraction

Figure 3.1 shows our method's abstract activities. They are not tied to any particular technology and can be instantiated to meet varied software development contexts.

The intention behind the first activity – *identify concerns* – is to encapsulate all steps needed to find information about the implementation of concerns. This includes locating reliable datasources and defining strategies to mine them. In the context of our research,



**Figure 3.1** Abstraction

we see third-party components’ metadata as a valuable datasource because developers are usually cautious and attentive when they inject components in their systems.

The second activity – *associate concerns with source code artifacts* – has the intention to determine which and how source code artifacts are linked to concerns. This is a necessary step to make further investigations possible (*e.g.*, the ones described in Chapter 5 and Appendix C).

The third activity of our method – *Export dataset for specialized analysis* – has the intention to support varied analysis scenarios. It must specialize our dataset to address specific traits regarding the association between concerns and source code artifacts. By executing this activity, we expect that researchers and practitioners end up with subsets of data to examine features and problems that they are interested in.

The fourth activity – *perform specialized analysis* – represents the processing of the subset of data produced by the third activity. This activity complements the previous one by including specific data manipulation routines to support different investigations.

The first and second activities include the steps that are necessary to mine concerns and associate them with source code artifacts. We complement these activities by adding two other ones (the third and the fourth activities) because we want our method to enable varied applications of the identification and analysis of concerns. For instance, (i) evaluating how deeply crosscutting concerns impact the modularization of systems (KICZALES, 1996)(HE; YE, 2015), and (ii) analyzing software architecture issues that are associated with concerns, *e.g.*, architecture erosion (ADAMS; JIANG; HASSAN, 2010). As we do not address any of these specific problems in this thesis, we are, at least, indicating that our method can support investigating them via exporting and analysis of concerns-related datasets.

### 3.1.2 Realization

The activities depicted in Figure 3.1 are the backbone of our method. They are independent of adopted software development approaches and technologies and can be specialized to address specific contexts. Figure 3.2 exhibits an example of realization of our method, *i.e.*, it shows how we derive our method’s realization from its abstract activities.

“Abstraction” encapsulates the same perspective shown in Figure 3.1. “Realization” comprises middle/low-level activities to automate the abstraction. We seek to automate the activities as much as possible to reduce the effort required by concerns identification. Next, we explain how we realized our method’s abstraction to make use of systems’ third-party components:

*Identify concerns*: the fulfillment of this activity relies on information that developers embed third-party components’ metadata. *Mining metadata* about components (Activity 1.1) enables the recovering of developers’ decisions regarding the implementation of concerns. We complement the mining of the metadata by *retrieving extra information about components* (Activity 1.2) from components’ repositories (explained in Section 2.3). We recommend that software development specialists review any dataset produced by the method and *reach a consensus about concerns* (Activity 1.3) to assure that the mined information is correct and precise. As a result, the method outputs a *categorized dataset* containing the association between concerns and the third-party components injected in systems;

*Associate concerns with source code artifacts*: we believe that conclusions about how concerns are implemented can be refined by analyzing their evolution through development. Consequently, we have defined an activity to *mine source historical data* (Activity 2.1) from projects’ Version Control Systems (VCS). It is also necessary to find a way to *associate the concerns extracted from components’ metadata with source code artifacts* (Activity 2.2). The association is vital to make causal investigations possible, *e.g.*, comprehending if the occurrence of a design problem can be traced back to the implementation of concerns. The method also *calculates concern-related metrics* (Activity 2.3) by processing Abstract Syntax Trees (AST) generated from systems’ source code. After this last activity, our method produces a *database* into which evolutionary information about concerns is stored;

*Export dataset for specialized analysis*: by mining concerns through the evolution of software projects, our method has the potential to generate a large dataset. We believe that it is advantageous to split the data into subsets to support specific studies. So, *defining export strategies* (Activity 3.1) represents the mental effort required to identify ways to extract such subsets and to develop strategies to *export* them as a *specialized dataset* (Activity 3.2);

*Perform specialized analysis*: we think that scripts are the best way to deal with the data mined from software projects. So, we added another activity to our method, *write*

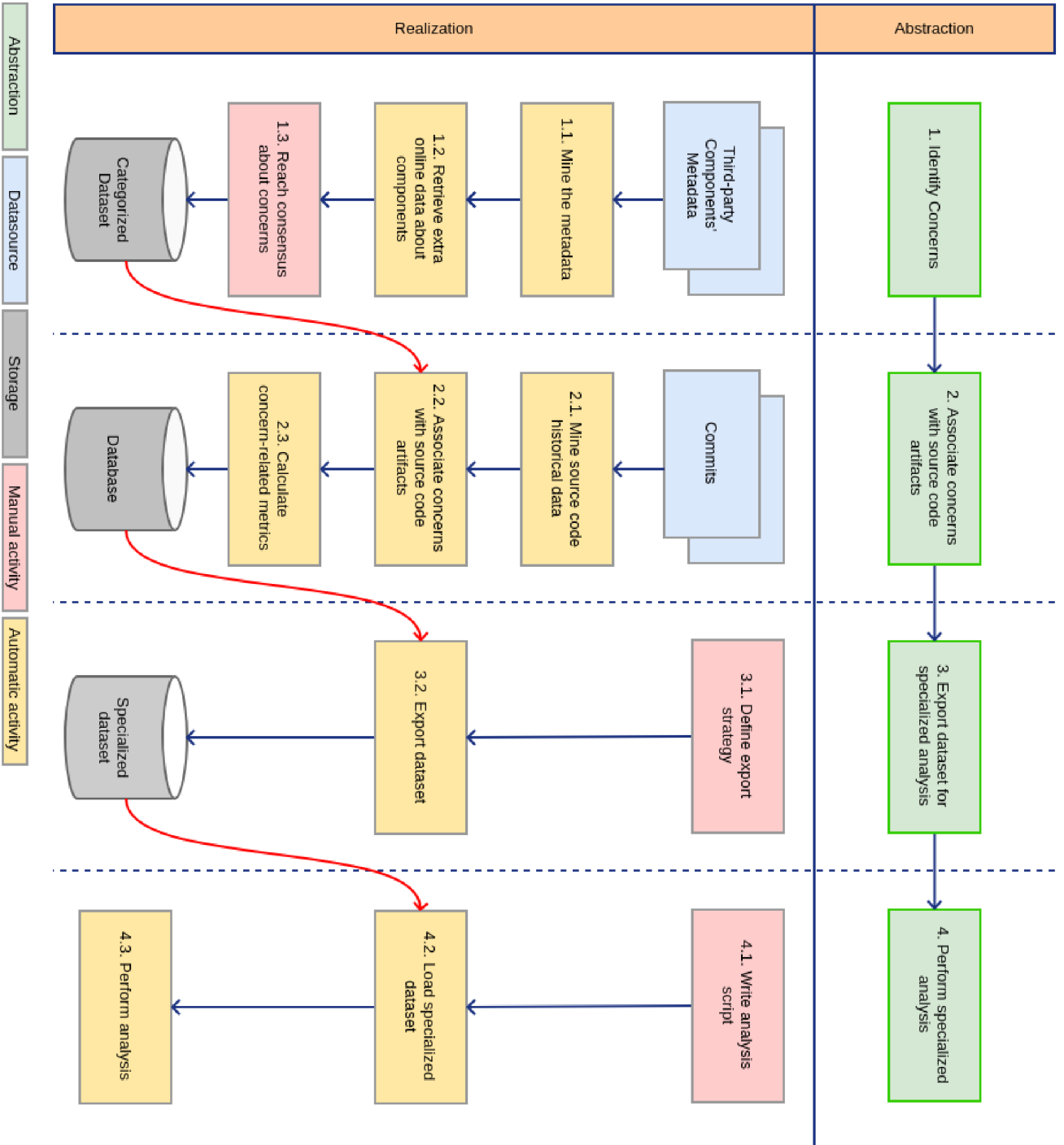


Figure 3.2 Realization – Adapted from (CARVALHO; NOVAIS; MENDONÇA, 2020)

*analysis scripts* (Activity 4.1). This activity must *load the specialized dataset* (Activity 4.2) and *perform analysis* regarding: (i) the presence of concerns in source code artifacts, (ii) their evolution through systems' development history and (iii) relationship with other software development phenomena (Activity 4.3).

It is important to emphasize that the first activity, *identify concerns*, is highly dependent on the availability of third-party components' metadata in software projects. Therefore, the more a project's developers rely on components-based programming the more our method is able to extract information about concerns. Conversely, other projects' developers may not inject many components into their source code. Such situations include developers failing to find adequate components to automatize systems' functionalities and having to code them themselves. In this case, our method will be less likely to spot and analyze the presence of concerns. However, we trust that there is always a certain level of dependability toward the use of third-party components in the making of every modern software. So, there will always be an opportunity for the use of our method.

### 3.1.3 Instantiating

We developed a tool, called Architectural Knowledge Suite (AKS), to instantiate our method. AKS mines software projects' concerns and their association with source code artifacts. It automates all activities that Figure 3.2 presents as "Automatic Activity". Few others are dependent on manual processing and analysis of data. They are identified in Figure 3.2 as "Manual Activity".

Figure 3.3 exhibits two possible ways to instantiate our method depending on the adopted development technology: java and javascript. In this chapter, we focus on how to instantiate our method through the perspective of java-oriented software projects. Appendix E showcases a different scenario in which we re-instantiate our method after replacing java with javascript. We suggest the following activities to process java-based systems' concerns:

*Identify concerns*: the fulfillment of this activity relies on information that developers embed in POM and Gradle files. Mining metadata about components from POM/Gradle files enables recovering developers' decisions regarding the implementation of concerns (Activity 1.1). We complement the mining of the metadata by retrieving information about components from MVNRepository (described in Section 2.3). The method uses MVNRepository to mine categories of components found in projects' POM/Gradle files (Activity 1.2). As MVNRepository does not provide a category for all components (*i.e.*, many components remain uncategorized) a manual classification of concerns is required (Activity 1.3). The execution of this last task generates a comprehensive dataset of categorized concerns. Section 3.1.3.1 contains more details about the identification and categorization of concerns;

*Associate concerns with source code artifacts*: the main goal of this activity is to determine which source code artifacts are affected by the implementation of concerns. For

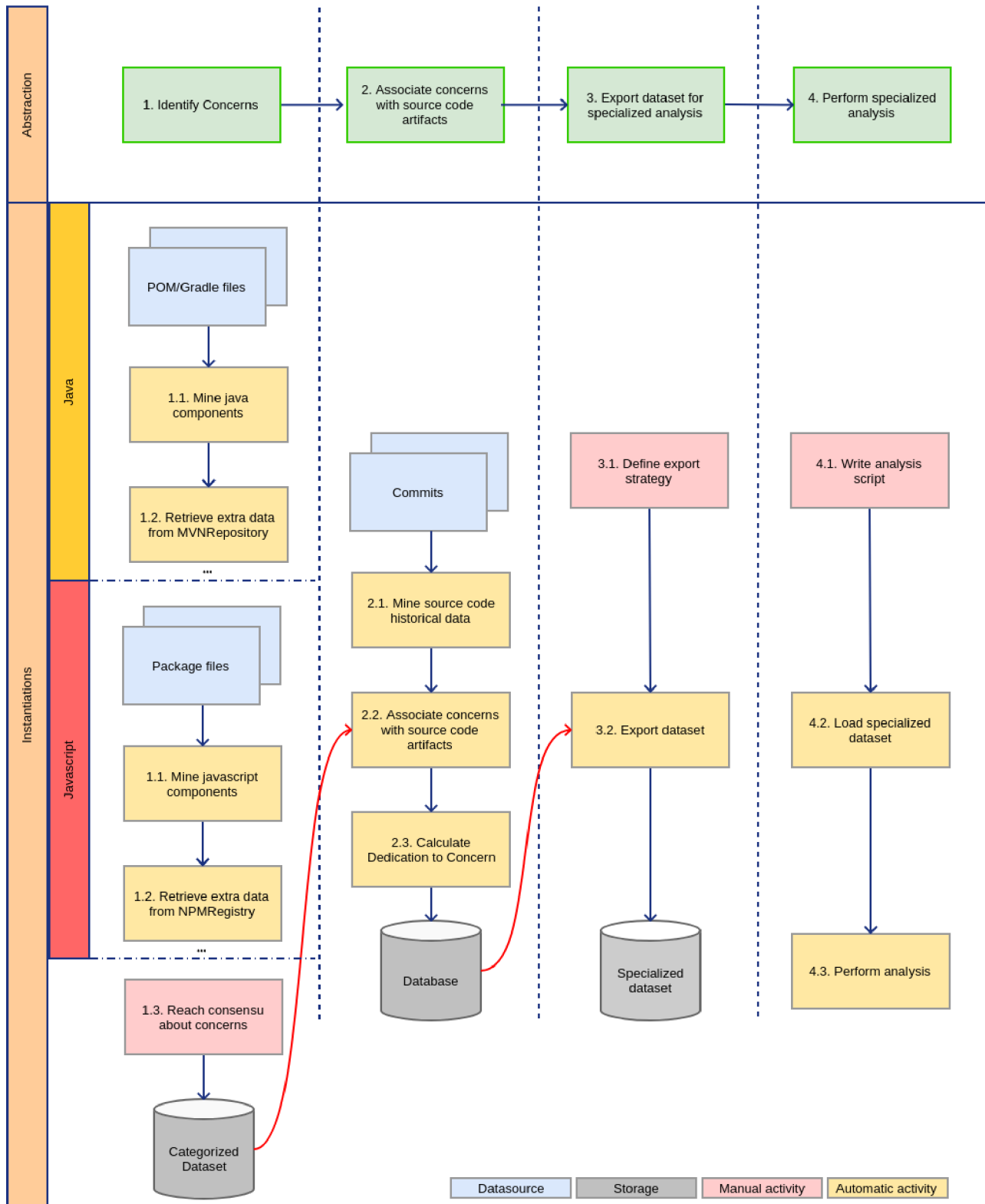


Figure 3.3 Method's Instances

instance, in java projects, this activity can associate the component, **dbunit**, with the “.java” files that inject it via import declarations (**import org.dbunit**) to automate tests. AKS uses mining routines implemented by Repository Miner (RM) to automate this activity. Calculating our metric, Dedication to Concern (DtC), is another important step carried out during the execution of this activity. In Section 3.1.4 we present more details about the characteristics and our use of DtC;

We want our method to support future studies that need to process the evolution of concerns. For instance, different analyses may require associating concerns with information found in source code repositories about (GIRBA; DUCASSE, 2006): reverse engineering, cost prediction, software aging, code styling and decaying. Thus, our method benefits from RM’s capability of mining historical data obtained from GIT repositories (Activities 2.1 and 2.2). After mining the historical data, our method determines the degree of association between concerns and artifacts using our DtC metric (Activity 2.3). Considering that the mining of concerns produces a large amount of data, we rely on MongoDB non-relational database<sup>1</sup> for storage;

*Export dataset for specialized analysis:* our tool can expose the information mined from software projects in a more concise and reusable way: as a comma-separated-value (CSV) dataset. We believe that exporting the dataset as CSV files maximizes reusability. This type of file can be processed by different tools (*e.g.*, spreadsheet editors) to automate investigations. This task requires the configuration of exporting strategies (Activity 3.1) to select data from the mined information and to generate and export a specialized dataset (Activity 3.2), *i.e.*, as the mining of concerns produces a large comprehensive database, it is necessary to extract excerpts of data to support specific studies;

*Perform specialized analysis:* we count on the use of R-language<sup>2</sup>-based scripts to load (Activity 4.2) and run the analysis (Activity 4.3) on the dataset generated by Activity 3. Externalizing the analysis as scripts favors the expansion of our approach to consider investigations other than the ones described in this work. This means other researchers/practitioners can reuse our dataset by writing their own scripts.

### 3.1.3.1 Identifying Java Projects’ Concerns

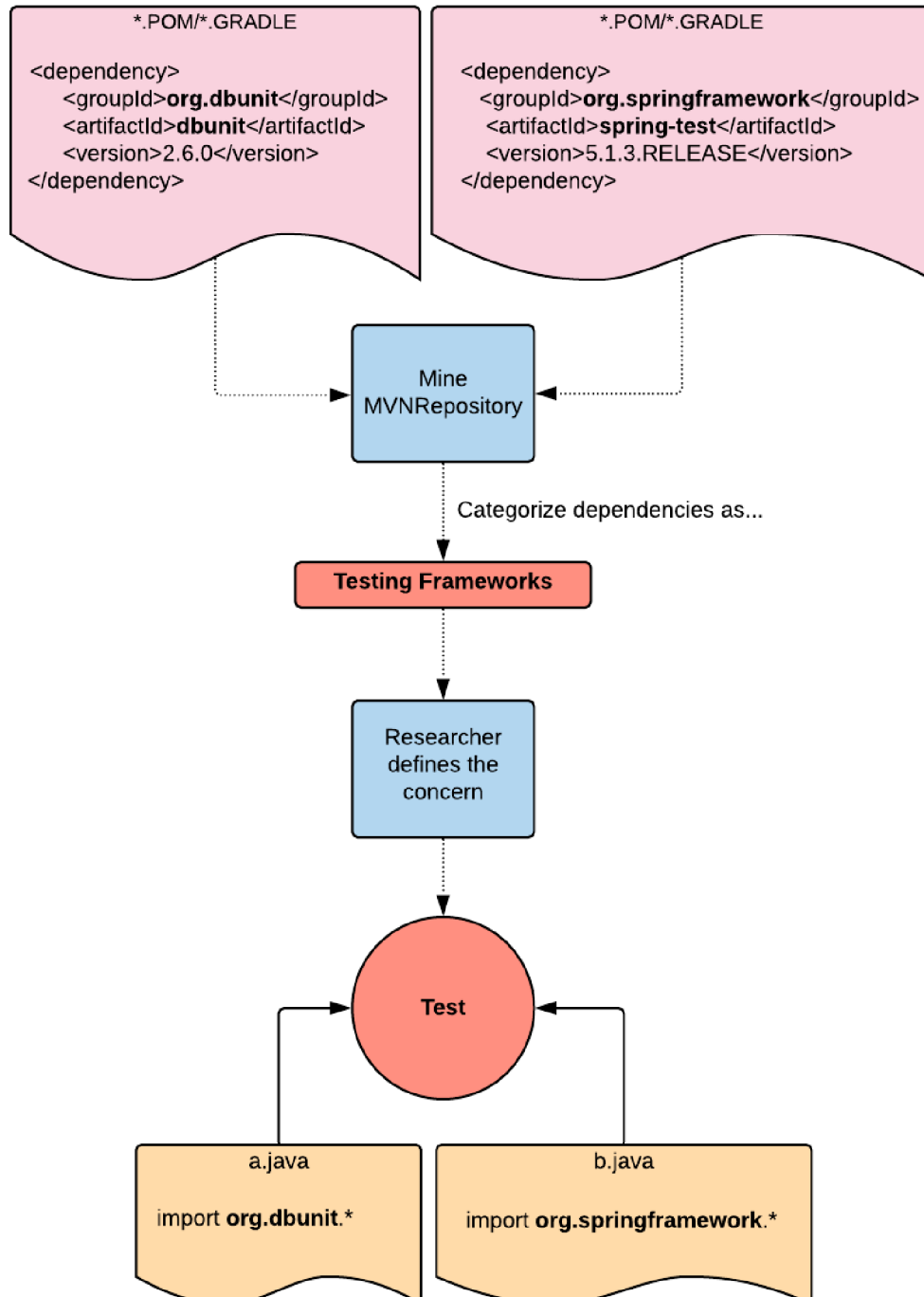
Figure 3.4 contains an example that shows how our tool, AKS, automates the mining of concerns from third-party components metadata. Suppose that developers added the components, **dbunit** and **spring-test**, to the POM files of two software projects. AKS is capable of processing the POM files to mine the components’ ids: groupId and artifactId. As the ids ensure each component is uniquely and unambiguously identified, AKS uses them to retrieve metadata from MVNRepository. The tool navigates to two distinct URLs obtained from the combination of components’ group and artifact ids: <https://mvnrepository.com/artifact/org.dbunit/dbunit> and <https://mvnrepository.com/>

---

<sup>1</sup><https://www.mongodb.com/>

<sup>2</sup><https://www.r-project.org/>

*artifact/org.springframework/spring-test*. By parsing the response from MVNRepository, AKS finds out that “Testing Frameworks” is how the repository has categorized the components. Then, we manually fill the most adequate concern to represent “Testing Frameworks”: “Test”.



**Figure 3.4** Mining of Concerns (CARVALHO; NOVAIS; MENDONÇA, 2020)

The manual classification of concerns can be seen as detrimental to the quality and



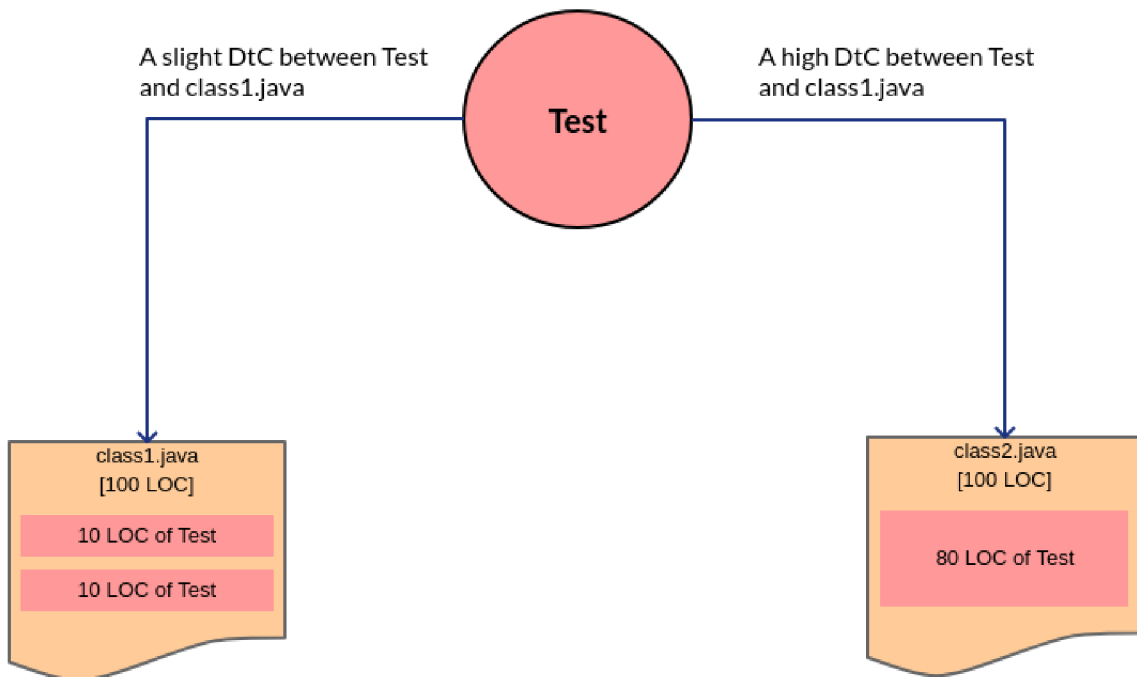
applicability of our method. However, we have prepared AKS to minimize the impact of this limitation by extracting as much extra data about components from MVNRespository as possible. The data (components’ descriptions, categories and keywords) can help researchers to precisely identify concerns. We have made the dataset of our studies available for reuse and replication after two reviewers assured their correctness (more information in Section 6.2). Both reviewers have academic and professional experience and consensually filled the missing categories and concerns after consulting the data obtained from MVNRespository and checking each component’s web sites, wikis, and other sources of information.

AKS associates source code artifacts with concerns by examining the list of import declarations in “.java” files. In Figure 3.4, file “a.java” imports **dbunit** and “b.java” imports **spring-test**. Then, AKS associates the “Test” concern with both artifacts.

### 3.1.4 Dedication to Concern

Dedication to Concern (DtC) is a metric we created in this thesis. We define DtC as the degree to which a source code artifact represents the implementation of a concern. This section presents an illustrative example to showcase how we conceptualized it.

DtC materializes the following rationale: given an artifact (*e.g.*, a “.java” file of a java-based program), the fewer concerns it contains the more dedicated to implement a specific system’s feature or requirement it is. On the other hand, more concerns decreases its dedication. Figure 3.5 illustrates how our method takes advantage of DtC to determine the strength of association between concerns and source code artifacts.



**Figure 3.5** Dedication To Concern (DtC)

Two source code artifacts of a java-based program, “class1.java” and “class2.java”, have the same amount of Lines of Code (100 LOC). Developers embedded the “Test” concern in the two of them. However, “class1.java” is not entirely dedicated to implement test routines. In “class1.java”, “Test” spreads over relatively few lines of code (20 LOC). Oppositely, developers decided to focus “class2.java” on automating tests through many of its lines of code (80 LOC). Thus, comparatively, we can say that “class2.java” is more dedicated to the implementation of “Test” than “class1.java”.

Among potential benefits of DtC, we can mention the possibility of developers focusing on slight associations between artifacts and concerns, *e.g.*, the association between “class1.java” and the “Test” concern exhibited in Figure 3.5. They may feel like applying refactoring strategies to revert the situation and isolate concerns in dedicated artifacts to achieve better modularity (MARCUS; POSHYVANYK; FERENC, 2008). In this case, changing “class1.java” to make it more dedicated to perform tests.

Next, we describe how to instantiate our method to identify concerns from java-based software projects. We also explain how we calculate DtC and associate it with source code artifacts.

#### 3.1.4.1 Measuring Object-Oriented Projects’ Dedication to Concern

Using import declarations to associate concerns with source artifacts does not suffice for a deeper analysis of how concerns impact information systems. The main problem is: importing a component does not guarantee that the component’s modules/classes are extensively used by an artifact’s class(es) and methods. We soon found situations in which the use of certain components was too diluted through artifacts’ lines of source code. We also found extreme cases in which no line of code referred to imported components. Developers may consider these situations uninteresting and having too little effect on the implementation of concerns. As a consequence, they might avoid analyzing them or rank them down to a low-priority category of impact. We then decided to measure the DtC of concerns to manage these situations.

We defined metrics and developed measurement strategies to make the categorization of DtC possible. We based the metrics on the elements that we can extract from object-oriented source code artifacts: import declarations, the use of parameters and variables by classes’ methods, and their relationship with concerns extracted from third-party components’ metadata. We defined the metrics, their thresholds and measurement approach based on our observations of source code snippets. After we reached a consensus about how to calculate DtC, we empowered AKS with routines to automate the processing of systems’ source code artifacts. We propose measuring DtC against a three-factor qualitative scale. By manually analyzing the source code of software projects, we noticed that we could group source code artifacts under three distinct categories of DtC regarding their relationships with concerns: slight, moderate, and high.

We consulted related literature to reuse metrics, but not all available ones suited us. For instance, the Number of Components (NOC) metric (ABILIO et al., 2015)(NUÑEZ-VARELA et al., 2017b) looked applicable, but its definition does not completely reflect our intention (ABILIO et al., 2016): NOC counts how many components (constants/re-

finements) are necessary to implement a feature. We are not sure if we can apply both concepts, “concern” and “feature”, interchangeably.

The Program Element Contribution (CONT) metric counts a program’s number of lines of code associated with concerns, but its definition (EADDY et al., 2008) does not include lines outside classes, *e.g.*, package declarations and imports. However, import declarations are vital for our method as they are used to find the artifacts that use/implement concerns.

Concern Diffusion over Components (CDC) and Concern Diffusion over Operations (CDO) (SANT’ANNA et al., 2003) look suitable. CDC counts the number of primary components whose main purpose is to contribute to the implementation of a concern. CDO counts the number of primary operations whose main purpose is to contribute to the implementation of a concern. According to their definitions, respectively, the metrics seem to accumulate the total number of components (classes or aspects<sup>3</sup>) and operations (*e.g.*, constructors, methods) related to a concern throughout a codebase. The problem is: we need to determine the dedication of one source code artifact to a concern per measurement. In other words, our measurement strategy must take an artifact “A” and a concern “C” as inputs and determine the strength of the association between the two. CDC and CDO take a concern “C” as input and return to total number of components and operations that refer to “C”. So, CDC and CDO does not completely fit in what we need to measure and how to measure it.

As we could not find an adequate set of metrics to take full advantage of the source code elements that we wanted to measure, we defined the metrics described in Table 3.1.

We applied a simple percentage ratio (DAWSON; O’NEILL, 2003) to calculate ICD and MD. This enabled us to fit both metrics’ numeric values into the three categories of dedication to concern that our method can measure (slightly/moderately/highly dedicated). We embedded functionalities in our tool to automate the execution of the following rule, which represents our rationale regarding the measurement of DtC:

$$\text{DtC}(A, C)^4 = \begin{cases} \text{SLIGHT} & \text{ICD}(A, C) = \text{SLIGHT} \vee (\text{ICD}(A, C) \in \\ & (\text{MODERATE}, \text{HIGH}) \\ & \wedge \text{MD}(A, C) = \text{SLIGHT}) \\ \text{MODERATE} & \text{ICD}(A, C) \in (\text{MODERATE}, \text{HIGH}) \wedge \\ & \text{MD}(A, C) = \text{MODERATE} \\ \text{HIGH} & \text{ICD}(A, C) \in (\text{MODERATE}, \text{HIGH}) \wedge \\ & \text{MD}(A, C) = \text{HIGH}, \end{cases}$$

where the **DtC** of an artifact **A** as it implements a concern **C** can be categorized as: (i) slight if the imported components’ dedication **ICD** is slight; (ii) slight if **ICD** is either moderate or high, and the methods’ dedication **MD** is slight; (iii) moderate if **ICD** is either moderate or high, and **MD** is moderate; and (iv) high if **ICD** is either moderate

<sup>3</sup>In the context of Aspect-Oriented Software Engineering/Development

<sup>4</sup> $\in = \text{contained}(in)$ ,  $\vee = \text{or}$ ,  $\wedge = \text{and}$

**Table 3.1** Dedication to Concern’s Metrics

<b>Metric</b>	<b>Description</b>	<b>Purpose</b>
NOI	Number Of Imports	NOI counts a source code artifact’s total number of imports
NOIC	Number Of Imported Concerns	NOIC counts the total number of imported components that are associated with a concern
NOM <sup>a</sup>	Number of Methods	Total number of methods of a source code artifact
NOR <sup>b</sup>	Number Of References	NOR counts the total number of methods that reference a component/concern
ICD	Imported Components’ Dedication	ICD is an indirect metric obtained from simple percentage ratio between NOI and NOIC (NOIC/NOI) and has the purpose of measuring how strong/weak is the relationship between an artifact and a concern considering the imported components
MD	Methods’ Dedication	MD measures methods’ dedication regarding a specific concern. We calculate MD as a simple percentage ratio between NOM and NOR (NOR/NOM)
DtC <sup>b</sup>	Dedication to Concern	DtC outputs the dedication of a source code artifact from values obtained from ICD and MD

<sup>a</sup> from (OLIVEIRA; VALENTE; LIMA, 2014)(NUÑEZ-VARELA et al., 2017a)

<sup>b</sup> similar to Concern Diffusion over Operations (CDO) (SANT’ANNA et al., 2003), except that our metric applies to methods of only one class per measurement <sup>c</sup> similar to Concern Diffusion over Components (CDC) (SANT’ANNA et al., 2003), except that our metric applies to only one class per measurement

or high, and **MD** is high.

We adopted the following as the thresholds for slight, moderate, and high: (i) slight is any value equal or below 0.3; (ii) moderate is any value greater than 0.3 and equal or below 0.6; and (iii) high is any value from 0.6 to 1.0. We obtained the mentioned values from a principle related to the definition of interval scales where the distance between adjacent elements (of a scale) must be constant and equidistant (BÖHME; FREILING, 2008)(AINI; ZULIANA; SANTOSO, 2018). Additionally, interval scales can be converted

to nominal by being cut at breakpoints and assigning the resulting slices of data to categories (BÖHME; FREILING, 2008), as the ones that we associated with DtC (slight, moderate and high).

We believe that the rule, metrics and thresholds that we presented through this section suffice for measuring DtC. However, we recognize that we may have introduced a bias in our research by not conducting side-studies to validate them. We include ourselves in the group of authors that defined metrics according to their experience. As a consequence, it may be difficult to reproduce or generalize our results (ALVES; YPMA; VISSER, 2010)(OLIVEIRA; VALENTE; LIMA, 2014). While we do not discard the evaluation of our metrics and their thresholds as mandatory, we tried our best to reduce the bias of our guesswork. At least, we followed the recommendation that defining metrics’ thresholds should be achieved by measuring data from a representative set of systems (ALVES; YPMA; VISSER, 2010). We accomplished this by investigating the evolution of software projects, as described in Chapters 4 and 5, and Appendixes C and E. Additionally, we took measures to ensure that our method, tool, dataset, and analysis scripts can be reviewed and modified to accept metrics and thresholds other than the ones that we have currently defined and applied.

## 3.2 APPLYING OUR DTC METRIC (A WORKED EXAMPLE)

Now, we exemplify how our tool uses our method’s metrics and rule to measure DtC. We base our explanations on examples taken from real software projects’ POM and source code files. We selected the projects from a specific domain of software: non-relational databases. Table 3.2 describes the databases.

**Table 3.2** Non-relational Databases

Database	Description
JanusGraph <sup>a</sup>	Highly scalable graph database
Neo4J <sup>b</sup>	High-performance graph store with the features expected from a robust database
KairosDb <sup>c</sup>	Fast distributed scalable time series database written on top of Cassandra

<sup>a</sup> <https://github.com/JanusGraph/janusgraph>

<sup>b</sup> <https://github.com/neo4j/neo4j>

<sup>c</sup> <https://github.com/kairosdb/kairosdb>

### 3.2.1 Measuring a High DtC

The first category of DtC, high, includes cases as the one exemplified in Listing 3.1. By adding **junit**’s import declarations (line 3 to 7, highlighted in yellow), developers connected the artifact to a test component. Consequently, the imports tie the artifact with a concern: “Test”. Additionally, we consider that the artifact is extensively dedicated

to the implementation of this concern because the `DataCacheTest` class encapsulates 2 methods (in gray), `test_isCached` (line 13) and `test_uniqueCache` (line 29), and each method encloses references to classes imported from `junit` (`Test`, `TestCase`, `Assert`). Therefore, we consider that the artifact is **highly dedicated** to implement the “Test” concern.

**Listing 3.1** Highly Dedicated Artifact

```

1 package org.kairosdb.datastore.cassandra;
2
3 import static import org.junit.Test;
4
5 import static junit.framework.TestCase.assertTrue;
6 import static org.junit.Assert.assertNotNull;
7 import static org.junit.Assert.assertNull;
8
9 public class DataCacheTest {
10     ...
11
12     @Test
13     public void test_isCached() {
14         DataCache<String> cache = \textbf{new} DataCache<String>(3);
15
16         assertNull(cache.cacheItem("one"));
17         assertNull(cache.cacheItem("two"));
18         assertNull(cache.cacheItem("three"));
19
20         assertNotNull(cache.cacheItem("one")); // This puts 'one' as the
21             newest
22         assertNull(cache.cacheItem("four")); // This should boot out '
23             two'
24         assertNull(cache.cacheItem("two")); // Should have booted 'three
25             ,
26         assertNotNull(cache.cacheItem("one"));
27         assertNull(cache.cacheItem("three")); // Should have booted '
28             four'
29         assertNull(cache.cacheItem("one"));
30     }
31
32     @Test
33     public void test_uniqueCache() {
34         TestObject td1 = new TestObject("td1");
35         TestObject td2 = new TestObject("td2");
36         TestObject td3 = new TestObject("td3");
37
38         DataCache<TestObject> cache = new DataCache<TestObject>(10);
39
40         cache.cacheItem(td1);
41         cache.cacheItem(td2);
42         cache.cacheItem(td3);
43     }
44 }

```

```

40     TestObject ret = cache.cacheItem(new TestObject("td1"));
41     assertTrue(td1 == ret);
42
43     ret = cache.cacheItem(new TestObject("td2"));
44     assertTrue(td2 == ret);
45
46     ret = cache.cacheItem(new TestObject("td3"));
47     assertTrue(td3 == ret);
48 }
49 }

```

Table 3.3 describes how AKS calculates the metrics for the artifact exhibited in Listing 3.1 and determine its DtC considering the “Test” concern.

**Table 3.3** Measuring a Highly Dedicated Artifact

Metric	Value	Measurement Strategy
NOI	4	Extracted from the artifact’s list of imports
NOIC	4	Calculated from the imports that implement the “Test” concern
NOM	2	As the total number of methods
NOR	2	Because two methods refer to classes imported from the “Test” component
ICD	1.0	We obtain 100% of dedication, as all imported classes are associated with the ‘Test’ concern
MD	1.0	Another 100% of dedication because all methods declare at least one class associated with the ‘Test’ concern
DtC	High	Processing ICD and MD through the rule reveals that the artifact is highly dedicated to implement ‘Test’

### 3.2.2 Measuring a Moderate DtC

Other artifacts do not focus on the implementation of a single concern. They tend to comprise import declarations which insert a relatively small set of components. For example, the source code in Listing 3.2 reveals that two different concerns are in play: “Test”, as it imports **junit** (line 3 to 5) and “Logging”, from the importing of **slf4j** (lines 7 and 8). Regarding the “Test” concern, developers associated 3 out of 9 artifact’s methods with the implementation of testing routines (lines 64, 77, and 90). We categorize the artifact as being moderately dedicated to implement “Test”.

**Listing 3.2** Moderately Dedicated Artifact

```

1 package org.janusgraph.testutil;
2
3 import org.junit.Test;

```

```

4  import static org.junit.Assert.assertEquals;
5  import static org.junit.Assert.assertTrue;
6
7  import org.slf4j.Logger;
8  import org.slf4j.LoggerFactory;
9
10 public class RandomGenerator {
11
12     private static final Logger log = LoggerFactory.getLogger(
13         RandomGenerator.class);
14
15     private static final int standardLower = 7;
16     private static final int standardUpper = 21;
17
18     public static String [] randomStrings(int number) {
19         return randomStrings(number, standardLower, standardUpper);
20     }
21
22     public static String [] randomStrings(int number, int lowerLen, int
23         upperLen) {
24         String [] ret = new String [number];
25         for (int i = 0; i < number; i++) ret [i] = randomString(lowerLen,
26             upperLen);
27         return ret;
28     }
29
30     public static String randomString() {
31         return randomString(standardLower, standardUpper);
32     }
33
34     public static String randomString(int lowerLen, int upperLen) {
35         assert lowerLen > 0 && upperLen >= lowerLen;
36         int length = randomInt(lowerLen, upperLen);
37         StringBuilder s = new StringBuilder();
38         for (int i = 0; i < length; i++) {
39             s.append((char) randomInt(97, 120));
40         }
41         return s.toString();
42     }
43
44     public static int randomInt(int lower, int upper) {
45         assert upper > lower;
46         int interval = upper - lower;
47         // Generate a random int on [lower, upper)
48         double rand = Math.floor(Math.random() * interval) + lower;
49         // Shouldn't happen
50         if (rand >= upper) rand = upper - 1;
51         // Cast and return
52         return (int) rand;
53     }
54 }

```



```

52  public static long randomLong(long lower, long upper) {
53      assert upper > lower;
54      long interval = upper - lower;
55      // Generate a random int on [lower, upper)
56      double rand = Math.floor(Math.random() * interval) + lower;
57      // Shouldn't happen
58      if (rand >= upper) rand = upper - 1;
59      // Cast and return
60      return (long) rand;
61  }
62
63  @Test
64  public void testRandomInt() {
65      long sum = 0;
66      int trials = 100000;
67      for (int i = 0; i < trials; i++) {
68          sum += randomInt(1, 101);
69      }
70      double avg = sum * 1.0 / trials;
71      double error = (5 / Math.pow(trials, 0.3));
72      // log.debug(error);
73      assertTrue(Math.abs(avg - 50.5) < error);
74  }
75
76  @Test
77  public void testRandomLong() {
78      long sum = 0;
79      int trials = 100000;
80      for (int i = 0; i < trials; i++) {
81          sum += randomLong(1, 101);
82      }
83      double avg = sum * 1.0 / trials;
84      double error = (5 / Math.pow(trials, 0.3));
85      // log.debug(error);
86      assertEquals(50.5, avg, error);
87  }
88
89  @Test
90  public void testRandomString() {
91      for (int i = 0; i < 20; i++) log.debug(randomString(5, 20));
92  }
93  }

```

Table 3.4 demonstrates how AKS calculates the metrics to classify the source code artifact in Listing 3.2 as moderately dedicated to implement the “Test” concern.

### 3.2.3 Measuring a Slight DtC

Listing 3.3 shows lines of code extracted from an artifact which imports too many different components. Considering the import declarations, only **junit** is associated with the “Test” concern (line 3 to 7), and 4 out of its 23 methods contain references to test classes

**Table 3.4** Measuring a Moderately Dedicated Artifact

Metric	Value	Measurement Strategy
NOI	5	Extracted from the artifact’s list of imports
NOIC	3	Calculated from the imports that implement the “Test” concern
NOM	9	As the total number of methods
NOR	3	Because three methods refer use classes imported from the “Test” component
ICD	0.6	A ratio between NOIC and NOI
MD	0.38	A ration between NOR and NOM
DtC	Moderate	Processing ICD and MD through the rule indicates that the artifact is moderately dedicated to implement ‘Test’

imported from **junit**<sup>5</sup> (lines 65, 74, 81, and 93). Consequently, regarding the “Test” concern, we determine that the artifact is **slightly dedicate** to implement it.

**Listing 3.3** slightly Dedicated Artifact

```

1 package org.neo4j.index.population;
2
3 import org.junit.After;
4 import org.junit.Before;
5 import org.junit.Test;
6 import static org.junit.Assert.assertEquals;
7 import static org.junit.Assert.assertNotNull;
8
9 import java.io.File;
10 import java.io.IOException;
11 import java.nio.file.Path;
12 import java.nio.file.Paths;
13 import java.util.ArrayList;
14 import java.util.List;
15 import java.util.concurrent.Callable;
16 import java.util.concurrent.ExecutionException;
17 import java.util.concurrent.ExecutorService;
18 import java.util.concurrent.Executors;
19 import java.util.concurrent.Future;
20 import java.util.concurrent.TimeUnit;
21 import java.util.concurrent.atomic.AtomicLong;
22 import java.util.function.LongSupplier;
23 import java.util.function.Supplier;

```

<sup>5</sup>We omitted the source code of all methods which are not related to the “Test” concern because the class is a large one and its exhibition was spreading over too many pages.

```

24
25 import org.neo4j.graphdb.GraphDatabaseService;
26 import org.neo4j.graphdb.Label;
27 import org.neo4j.graphdb.Node;
28 import org.neo4j.graphdb.Transaction;
29 import org.neo4j.graphdb.schema.ConstraintDefinition;
30 import org.neo4j.graphdb.schema.IndexDefinition;
31 import org.neo4j.graphdb.schema.Schema;
32 import org.neo4j.io.fs.FileUtils;
33 import org.neo4j.test.TestGraphDatabaseFactory;
34
35 import static org.apache.commons.lang3.SystemUtils.JAVA_IO_TMPDIR;
36 import static org.neo4j.helper.StressTestingHelper.fromEnv;
37
38 public class LucenePartitionedIndexStressTesting {
39     private static final String LABEL = "label";
40     private static final String PROPERTY_PREFIX = "property";
41     private static final String UNIQUE_PROPERTY_PREFIX = "
        uniqueProperty";
42
43     private static final int NUMBER_OF_PROPERTIES = 2;
44
45     private static final int NUMBER_OF_POPULATORS =
46         Integer.valueOf(
47             fromEnv(
48                 "LUCENE_INDEX_NUMBER_OF_POPULATORS",
49                 String.valueOf(Runtime.getRuntime().
                    availableProcessors() - 1)));
50     private static final int BATCH_SIZE =
51         Integer.valueOf(fromEnv("LUCENE_INDEX_POPULATION_BATCH_SIZE",
                    String.valueOf(10000)));
52
53     private static final long NUMBER_OF_NODES =
54         Long.valueOf(fromEnv("LUCENE_PARTITIONED_INDEX_NUMBER_OF_NODES",
                    String.valueOf(100000)));
55     private static final String WORK_DIRECTORY =
56         fromEnv("LUCENE_PARTITIONED_INDEX_WORKING_DIRECTORY",
                    JAVA_IO_TMPDIR);
57     private static final int WAIT_DURATION_MINUTES =
58         Integer.valueOf(fromEnv("
                    LUCENE_PARTITIONED_INDEX_WAIT_TILL_ONLINE", String.valueOf
                    (30)));
59
60     private ExecutorService populators;
61     private GraphDatabaseService db;
62     private File storeDir;
63
64     @Before
65     public void setUp() throws IOException {
66         storeDir = prepareStoreDir();
67         System.out.println(String.format("Starting database at: %s",
                    storeDir));

```

```

68
69     populators = Executors.newFixedThreadPool(NUMBER_OF_POPULATORS);
70     db = new TestGraphDatabaseFactory().newEmbeddedDatabaseBuilder(
71         storeDir).newGraphDatabase();
72 }
73 @After
74 public void tearDown() throws IOException {
75     db.shutdown();
76     populators.shutdown();
77     FileUtils.deleteRecursively(storeDir);
78 }
79
80 @Test
81 public void indexCreationStressTest() throws Exception {
82     createIndexes();
83     createUniqueIndexes();
84     PopulationResult populationResult = populateDatabase();
85     findLastTrackedNodesByLabelAndProperties(db, populationResult);
86     dropAllIndexes();
87
88     createUniqueIndexes();
89     createIndexes();
90     findLastTrackedNodesByLabelAndProperties(db, populationResult);
91 }
92
93 private void findLastTrackedNodesByLabelAndProperties(
94     GraphDatabaseService db, PopulationResult populationResult) {
95     try (Transaction ignored = db.beginTx()) {
96         Node nodeByUniqueStringProperty =
97             db.findNode(
98                 Label.label(LABEL), getUniqueStringProperty(),
99                 populationResult.maxPropertyId + "");
100        Node nodeByStringProperty =
101            db.findNode(Label.label(LABEL), getStringProperty(),
102                populationResult.maxPropertyId + "");
103        assertNotNull("Should find last inserted node",
104            nodeByStringProperty);
105        assertEquals(
106            "Both nodes should be the same last inserted node",
107            nodeByStringProperty,
108            nodeByUniqueStringProperty);
109
110        Node nodeByUniqueLongProperty =
111            db.findNode(Label.label(LABEL), getUniqueLongProperty(),
112                populationResult.maxPropertyId);
113        Node nodeByLongProperty =
114            db.findNode(Label.label(LABEL), getLongProperty(),
115                populationResult.maxPropertyId);
116        assertNotNull("Should find last inserted node",
117            nodeByLongProperty);

```

```
112     assertEquals(
113         "Both nodes should be the same last inserted node",
114         nodeByLongProperty,
115         nodeByUniqueLongProperty);
116     }
117 }
118
119 private void dropAllIndexes() {
120     ...
121 }
122
123 private void createIndexes() {
124     ...
125 }
126
127 private void createUniqueIndexes() {
128     createIndexes(true);
129 }
130
131 private void createIndexes(boolean unique) {
132     ...
133 }
134
135 private PopulationResult populateDatabase() throws
136     ExecutionException, InterruptedException {
137     ...
138 }
139 private File prepareStoreDir() throws IOException {
140     ...
141 }
142
143 private PopulationResult populateDb(GraphDatabaseService db)
144     throws ExecutionException, InterruptedException {
145     ...
146 }
147
148 private void createAndWaitForIndexes(boolean unique) {
149     ...
150 }
151
152 private void createUniqueConstraint(int index) {
153     ...
154 }
155
156 private void createIndex(int index) {
157     ...
158 }
159
160 private void awaitIndexesOnline(GraphDatabaseService db) {
161     ...
162 }
163
```

```

164     private static String getLongProperty() {
165         ...
166     }
167
168     private static String getStringProperty() {
169         ...
170     }
171
172     private static String getUniqueLongProperty() {
173         ...
174     }
175
176     private static String getUniqueStringProperty() {
177         ...
178     }
179
180     private static class SequentialStringSupplier implements Supplier<
181         String> {
182         ...
183     }
184     private static class SequentialLongSupplier implements
185         LongSupplier {
186         ...
187     }
188     private static class Populator implements Callable<Long> {
189         ...
190     }
191
192     private class PopulationResult {
193         ...
194     }
195 }

```

In Table 3.5 we show how our method calculates the metrics to determine the dedication of the source code artifact exhibited in Listing 3.3. Considering the “Test” concern, AKS determines that the artifact is slightly dedicated.

In summary, our method and AKS have the ability to classify the DtC between concerns and source code artifacts in one of the following categories: highly dedicated, moderately dedicated, and slightly dedicated.

### 3.3 DISCUSSION

In this chapter, we described our method to support the identification and analysis of concerns. We designed it as a way to circumvent some limitations: (i) the tendency of the manual identification of concerns to be fatiguing and error-prone, (ii) the inadequacy of available automated approaches, and (iii) the lack and imprecision of Software Requirement Documents (SRDs) and Software Architecture Documents (SADs). We do not discredit any of these strategies as approaches to enable the analysis of the impact of concerns. However, we saw a gap in the way how they spot concerns and allow the mining of

**Table 3.5** Measuring a Slightly Dedicated Artifact

Metric	Value	Measurement Strategy
NOI	16	Extracted from the artifact’s list of imports
NOIC	5	Calculated from the imports that implement the “Test” concern
NOM	23	As the total number of methods
NOR	4	As four methods refer to at least one class imported from the “Test” component
ICD	0.31	A ratio between NOIC and NOI
MD	0.17	A ration between NOR and NOM
DtC	Slight	a result obtained from the processing of ICD and MD through the rule

evolutionary data regarding their implementation. So, we based our method on the use of POM and Gradle files as sources of concern-related information. Another key aspect of our method is that it can determine the strength of association between concerns and source code artifacts (DtC). A varying DtC can impact the way how developers evaluate concerns.

Although we assure that our method is adequate enough to analyze the impact of concerns on software projects, we must point out some limitations. Ideally, the method should grant the full automation of the analysis process because we aim it at evaluating the impact of concerns through the history of software projects. This can generate a dataset so extensive that its manual processing can become burdensome. As an example of a limitation, we can mention the need for filling missing components’ categories and concerns after the mining of software projects.

We recognize another limitation in our use of components’ metadata: not all concerns are implemented with the help of third-party components. Taking the source code exhibited in Listing 3.3 as an example, it is possible to notice that some of the imports (line 25 to 33) links the artifact with internal classes of **Neo4j**. This indicates that Neo4j’s developers implemented some concerns with no help of components. Consequently, Neo4j’s POM files may not contain a full set of information about its concerns. Considering that our method relies on POM/Gradle files, it cannot identify all Neo4j’s concerns.

### 3.4 RELATED WORK

We highlight the following as studies that describe techniques related to the automatic/semi-automatic identification concerns:

Robillard and Murphy (2002) proposed a way to generate graphs from key abstract structures used to implement concerns. They believe that this is more effective in documenting and analyzing concerns in comparison to lines of source code. They even created a tool to support their method: Feature Exploration and Analysis Tool (FEAT). FEAT

has the purpose of supporting maintenance tasks by visualizing concern-based graphs and querying some metrics, *e.g.*, fan-in, fan-out. Similarly to our method, they rely on structural aspects mined from the source code to find concerns.

Poruban and Nasal (2014) saw an opportunity in projecting multiple concerns over the same pieces of source code. Projections can help developers to perceive which concerns overlap with each other regarding a system's module (*e.g.*, a class in an object-oriented system). The Sieve Source Code Editor (SSCE) is the tool that the authors of the study developed to support the visualization of overlapped concerns. Our method also allows the analysis of multiple associations between source code artifacts and concerns. This comes from our approach to link artifacts' import declarations with concerns, *i.e.*, one artifact may contain more than one import declaration and this can link the artifact with many concerns.

Juhar and Vokorokos (2015) devised a way to determine which level of granularity most developers consider useful when dealing with concerns. With the help of their tool, Code Tagger (CT), they enabled software specialists to tag/mark source code fragments as concerns. Compared to AKS, CT provides a more fine-grained approach: it is capable of isolating fractions of methods' bodies of code (statements) to represent concerns, while AKS is limited to count the number of references to a component/concern found in methods. Although more precise, CT requires developers to manually associate concerns with a corresponding scope of statements. As the approach shown in Figure 3.3 is the most that our method can do to extract concerns from the source code of systems, favoring either CT or AKS means deciding on the level of mechanization that is desired from this type of tool.

He and Ye (2015) investigated if it is possible to identify concerns during requirements definition phases. They conceived a method based on goal models and on a two-state algorithm. The goal model is responsible for extracting relationships between different requirements' goals and the algorithm is used to automate the analysis of the relationships. The study focuses on applying the method to identify concerns from modeled aspects of systems.

Shaikh and Lee (2016) applied Aspect-Oriented Re-Engineering (AORE) to mine concerns from legacy systems to transform them into Aspect-Oriented applications. Their method requires the identification of concerns before refactoring systems' source code. Specifically, the authors targeted code smells (FOWLER; BECK, 1999) as elements to be turned into aspects. To achieve this, they applied a set of tools to find instances of code smells and to exploit Formal Concept Analysis (FCA) to group smelly modules that belong to the same concerns.

Nunez-Varela *et al.* (2017a) based a concern identification method on specialized information retrieval techniques. The techniques allow finding relevant information from a collection of documents containing unstructured text, *i.e.*, the source code is seen as unstructured text and the method can identify classes that contain core concerns by finding a Content Similarity Score (CSS) between modules. The paper does not state if a tool was created to automate the method.

Considering the aforementioned studies, we are the first ones to have ever addressed the possibility of extracting concerns from third-party components' metadata. We em-



phasize other advantages of our methods: (i) third-party components' metadata are frequently added to systems. This is a widespread good practice that makes several software projects available for investigating concerns; (ii) developers pay close attention to the injection of components, so the metadata tends to be precise; and (iii) as POM and Gradle files are important source code assets, they are often added to version control systems. This is advantageous as it enables examining the evolution of concerns.

### 3.5 CONCLUSION

In this chapter, we presented our method. It is composed of a set of activities from abstract to middle/low level ones. The decision to design it this way came from our perception that taking advantage of third-party components' metadata could be materialized to address different software development contexts. To showcase our method's flexibility, we describe a proof of concept in Appendix E. In Chapters 4 and 5, we identify and analyze concerns from java-oriented software projects, but in Appendix E we extend our method to deal with javascript applications. Additionally, we exemplify other types of software projects and programming languages (*e.g.*, python, PHP, ruby) that also contain third-party components' metadata, which may enable expanding the applicability of our method.

We did not find references about all metrics that we need to calculate the strength of the association between concerns and source code artifacts, or Dedication to Concern (DtC) as we call it. Therefore, we had to combine some few available metrics with new ones. This made us perceive that we had to count on the help of software development specialists to validate and refine our DtC's detection rule. Chapter 4 describes how we conducted an action research study (an approach that we presented in Section 2.7) with the purpose of validating and improving our method.



*In a room full of top software designers, if two agree on the same thing, that's a majority – Bill Curtis*

## STUDY I – AN ACTION RESEARCH STUDY TO EVALUATE OUR METHOD

In Chapter 3 we presented our method to support the identification and analyses of concerns. In order to validate and enhance the proposed method, we conducted an action research study.

From our perspective, we consider that action research studies are a good practical way of acquiring knowledge from software development specialists on how to adequately identify concerns and measure DtC. Thus, we must emphasize the following purpose behind this study: as we have unveiled a new way to extract concerns from software projects, we could not find answers for all questions that we had in related technical and theoretical literature. For instance, we associate the processing of Abstract Syntax Trees (AST) with the extraction of data from components' metadata. Does it suffice to spot concerns? To which degree does our method reflect developers' perception when they need to identify and process information about concerns? Consequently, we resorted to software developers' suggestions and opinions to enhance our method, tool, dataset and analysis scripts. This resonates with one of the purposes of action research studies: technology conception and tailoring with intense collaboration and changing through cyclical intervention (SANTOS; TRAVASSOS, 2011).

Santos and Travassos (2011) defined a template to report action research studies. Their template reflects the activities of the canonical template shown in Figure 2.4 (in Section 2.7). They proposed reporting action research studies as a sequence of stages: **Diagnostic, Planning, Actions, Evaluation and Analysis and Reflections and Learning**. Through next sub-sections, we describe how we instantiated the template.

### 4.1 DIAGNOSTIC

**Diagnostic** is composed of three sub-phases: Problem Description (PD), Project Context (PC) and Research Theme (RT). PD describes the problem faced. PC informs where the problem happens. RT summarizes the study to limit its scope. We presented our PD

through Chapter 1 and we summarize it here as:

*Software Documents, e.g., Software Requirements Documents (SRDs) and Software Architecture Documents (SADs), and available manual/automated approaches do not suffice for identifying and analyzing concerns*

We highlight the following as our study’s PC:

*We focus on Open Source Systems (OSS) as our main source of information about concerns. As the open source community has made many open source projects available in public repositories, we find it easier to base our studies on OSS. As any other type of software, OSS also lack adequate ways to spot concerns in their codebase*

We outline the following about our action research study’s RT:

*We want to identify concerns with the help of metadata, which developers add to software projects when they need to embed third-party components in their applications*

## 4.2 PLANNING

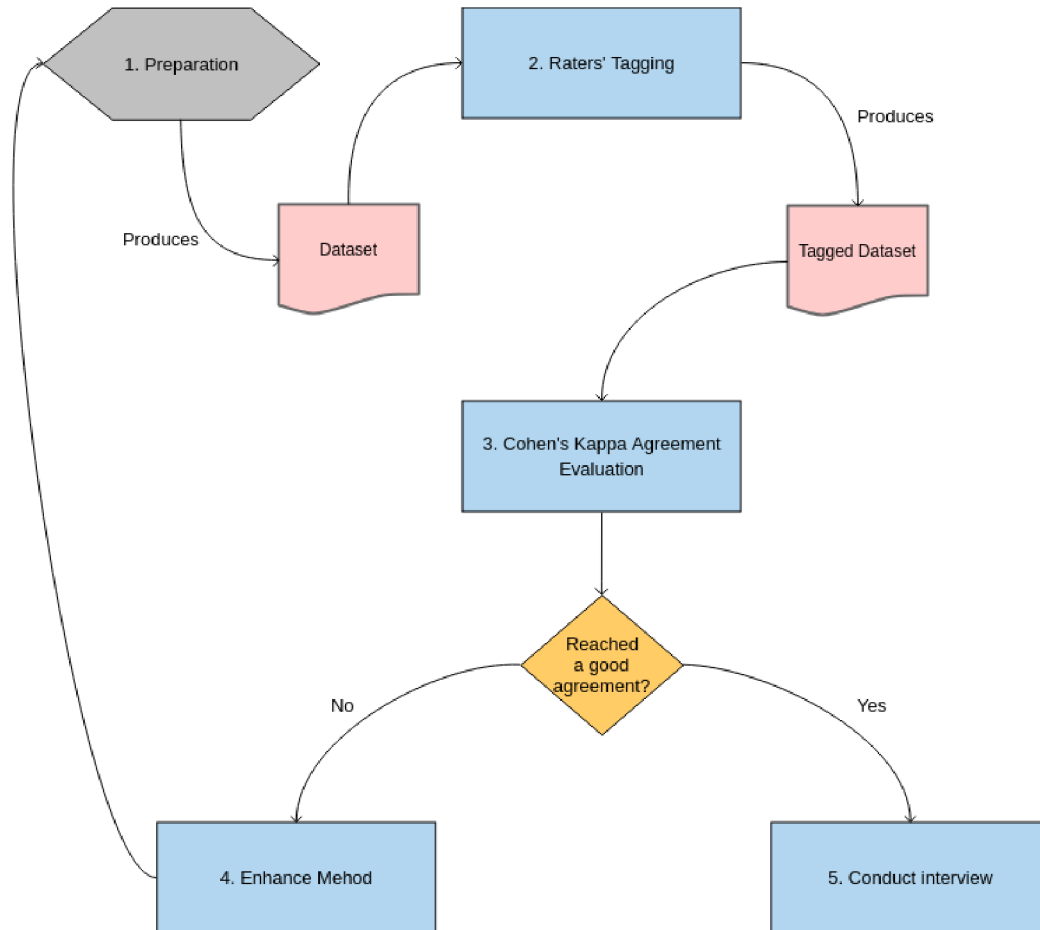
**Planning** is crucial to support research execution. It must include information about: (i) the technical aspects and literature surveys to ground the study (P1), (ii) controlled/pilot studies to determine the risks of using software technologies (P2) and (iii) the operational elements that are necessary to execute the research (P3). The set of theoretical and technical topics that we discussed in Chapter 2 stands for our P1. We already applied our method and tool in two previous studies (CARVALHO; NOVAIS; MENDONÇA, 2018)(CARVALHO; NOVAIS; MENDONÇA, 2020). Thus, we are confident AKS is robust enough to support new investigations (P2) and can provide us with a precise dataset to interact with software development specialists (P3).

## 4.3 ACTIONS

**Actions** are concerned with arranging study’s tasks and interventions in chronological order. Figure 4.1 shows how we organized our study’s actions.

The first action comprises all activities that we performed to prepare the study (1), *e.g.*, producing a dataset which should contain all the necessary information to help the raters to tag their agreement and disagreement. AKS is responsible for generating the dataset after mining and analyzing software projects. We sent the dataset to raters after instructing them about how to manipulate it. We processed the raters’ tagged datasets (2) using kappa coefficient to calculate the strength of agreement/disagreement (3). In case we reached no agreement, we took the opportunity to collect raters’ opinions about why they disagreed. We then used their opinions to enhance our method and tool (4). After this, we restarted the study. Otherwise, reaching an adequate strength of agreement finalized our evaluation and we conducted a semi-structured interview with the raters (5).

The interview had the purpose of discussing the results of the study and clarifying certain questions. Through the next sub-sections, we describe how we fulfilled these activities.



**Figure 4.1** Actions of our Action Research Study

### 4.3.1 Generating the Study's Dataset

The dataset should comprise a collection of samples to illustrate how our method finds and processes concerns. To achieve this, we used AKS to mine historical data from source code of open source software projects. Specifically, we focused on projects which were developed with the help of third-party components and the use of POM/Gradle files. We also decided to adopt “domain of software” as an inclusion strategy to select software projects<sup>1</sup>. Considering the two criteria, the second one (selecting projects according to their domains) is more restrictive. While POM and Gradle files have become a common presence in java projects, finding software projects related to specific areas of expertise and application is not always granted. Luckily, we were able to find interesting active projects under a domain we targeted: non-relational databases.

<sup>1</sup>Choosing “domain of software” as a strategy to select software projects is a consequence of one of our previous studies' findings. More information in Appendix C.

Table 4.1 contains the list of the software projects we analyzed. We extracted concerns from the source code of two types of non-relational databases domain: graph and time series databases. We mined 6 versions/releases from each database. **JanusGraph**, **Neo4j**, and **Titan** are graph-oriented Databases. **OpenTSDB**, **KairosDb**, and **Timely** are time series databases. The *Period* column informs the interval of historical data that AKS processed from the projects (six versions of each project are contained within the mentioned period). The *File* column informs the total number of source code files that AKS analyzed. After mining the software projects, AKS filled a database with historical information about concerns. The complete list of concerns can be found in Appendix A.

**Table 4.1** Non-relational Databases (CARVALHO; NOVAIS; MENDONÇA, 2020)

Domain	Project	Description	Period	Files
Graph	JanusGraph <sup>a</sup>	Highly scalable graph database	2017-04 - 2018-10	5657
	Neo4J <sup>b</sup>	High performance graph store with all the features expected from a robust database	2018-09 - 2018-12	26497
	Titan <sup>c</sup>	Database optimized for storing and querying large graphs	2012-06 - 2015-09	3570
Time Series	OpenTSDB <sup>d</sup>	Distributed, scalable TS database	2015-11 - 2018-12	1440
	KairosDb <sup>e</sup>	Fast distributed scalable TS database written on top of Cassandra	2015-11 - 2018-11	1884
	Heroic <sup>f</sup>	A scalable time series database based on Bigtable, Cassandra, and Elasticsearch	2016-06 - 2017-08	4258

<sup>a</sup> <https://github.com/JanusGraph/janusgraph>

<sup>b</sup> <https://github.com/neo4j/neo4j>

<sup>c</sup> <https://github.com/thinkaurelius/titan>

<sup>d</sup> <https://github.com/OpenTSDB/opentsdb>

<sup>e</sup> <https://github.com/kairosdb/kairosdb>

<sup>f</sup> <https://github.com/spotify/heroic>

Running this study required that we (i) instantiated our method to process java-based software projects and (ii) identified concerns implemented by such projects’ developers. We filled all missing concerns from the categories of components that ASK retrieved from MVNRepository. As explained in Section 3.1, our method and AKS are capable of generating a dataset for specialized analyses. We then implemented an exporting strategy to externalize concerns-related information in the form of a CSV dataset. We ensured that the dataset:

***Contained a variety of concerns mined from all the aforementioned databases:*** the agreement dataset<sup>2</sup> should include at least one concern found in each database. We

<sup>2</sup>From now on, we will use the terms “dataset” and “agreement dataset” interchangeably

wanted the raters to have a broad view of all possible combinations between source code artifacts and concerns;

***Included randomly selected samples from the variety of concerns-related information:*** this step is mandatory to increase the confidence of our findings while avoiding the bias of manual selection of the data;

***Granted a way to verify the relationship between concerns and source code artifacts:*** the exporting strategy linked the dataset with samples of source code. This means: at any moment, raters could check the projects' original source code files from which we mined concerns.

During each round of the study, the raters tagged a dataset in the format illustrated in Table 4.2<sup>3</sup>. The *Project* column informs the name of the software project. The *Concern* column indicates one specific concern to be analyzed. The list of imports that implement the concern is shown in the *Imports* column. The *Link to File* column allows the raters to verify the source code associated with the concern. The *DtC* column encapsulates the categorization of our Dedication to Concern metric. Raters could use the *Confirm?* column to either agree or disagree with our identification and categorization of concerns. We encouraged the raters to add their comments to the *Comment* column to inform us why they disagreed.

We populated the agreement dataset with 230 random samples from the historical data of the projects listed in Table 4.1. We calculated this number as a manner to guarantee a 5% error margin considering the total amount of selected samples:  $\cong 1500$ . The actual original dataset produced by AKS comprised a total of  $\cong 61500$  records.

Figure 4.2 shows how we produced the agreement dataset. As our method enables AKS to extract concerns from the source code of systems, we ended up with an extensive dataset ( $\cong 61500$  records). Then, we wrote a script to select random samples from it. We designed the script in a way that it could find a set of samples considering the most common concerns, *i.e.*, concerns that usually occur in all analyzed versions of the database software projects. This made the first reduction of the dataset possible. We reduced it down to  $\cong 1500$  records. To avoid fatiguing the raters we wrote a second script to select  $\cong 230$  items from the  $\cong 1500$  records.

### 4.3.2 Dataset Analysis Process

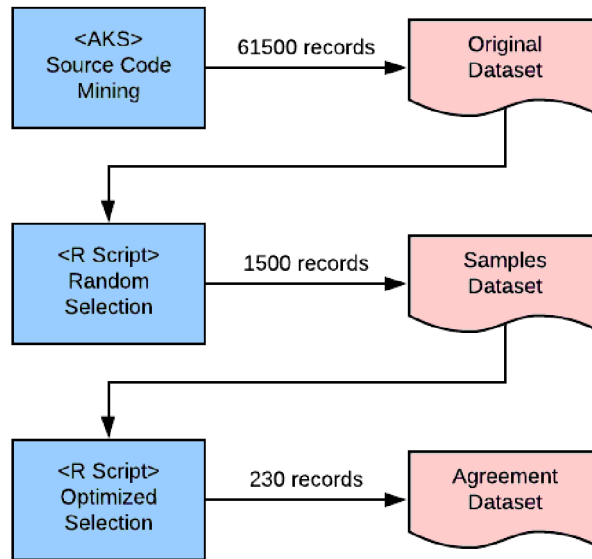
We recommended the following evaluation workflow to the raters: (i) the rater should check which concern he/she must evaluate, as informed in the *Concern* column; (ii) he/she should visualize the content of the source code as supplied by the *Link to File* column; (iii) he/she should verify the list of imported components that implement the concern (in the *Imports* column); (iv) by locating the concern's imported classes in the source code, he/she could either agree or disagree and enter YES (agree) or NO (disagree) in the *Confirm?* column after checking the categorization in the *DtC* column.

---

<sup>3</sup>We are showing actual values from one of our agreement spreadsheets.







**Figure 4.2** Dataset Reduction

We instructed the raters about the three categories of DtC and asked them to fit their opinions within our scale of slight, moderate, and high (as we discussed in Section 3.2). In other words, they were not free to fill their own categorization of DtC in the *DtC* column. However, they could add comments (in the *Comments* column) to inform how they would categorize the concerns in case they disagreed.

We did not impose a deadline for raters to tag and return the dataset. We saw this as a way to give them all time they needed to evaluate the dataset with attention.

### 4.3.3 The Raters

As a way to count on precise technical/theoretical opinions from specialists, we selected raters who met the following requirements:

1. The specialists should have experience in the development of software projects: this was important because we wanted to collect their opinions regarding our method in general and not only about the identified concerns. They could empower us with reliable opinions about different aspects of our method and help us to identify new ways to identify and analyze concerns;
2. Although we needed the specialists to have experience in software development, we did not limit the choice of candidates to ones with professional experience only: in other words, we also regarded having academic experience as a valuable aspect to select raters. We saw this as a way to complement specialists' practical opinions with theoretical viewpoints. Santos and Travasso (2011) consider that mixing professional and academic experience is vital for the relevance of scientific results. Ideally, the results must be useful and applicable in both industrial and academic environments.

One of the raters has been working as software developer for 14 years and he/she has a specialization in software engineering (*lato sensu*) and is currently enrolled in a master degree program (computer science).

The second rater worked for 10 years as a system analyst and has been teaching software engineering and development in a Brazilian public educational institute. He/she has a specialization in information systems engineering (*lato sensu*) and is also engaged in a master degree program.

We needed an extra rater during the third round of dataset tagging (more details in Section 4.4.3). He/she has a master degree in computer science and has professional experience too. For the last 8 years, the rater has been participating in software projects as a programmer and software architect and engineer.

All raters have experience in developing java-based software projects. This is important because all projects mentioned in Table 4.1 were written in Java.

We selected the raters via “convenience sampling” (KITCHENHAM; PFLEEGER, 2002). We invited the three raters from the same post-graduation program<sup>4</sup>. As the author of this work is also attending the program, it was convenient to select raters from a local group of students. According to Kitchenham and Pfleeger (2002), a problem may arise from this approach: people who are available and willing to participate may differ in important ways from those who are not willing. For instance, as being colleagues, they may tend to favor our method. However, throughout the phases of our study, we constantly reinforced to raters that their impartial opinions would be very important to enhance our method and tool. They should stick to their beliefs regarding the identification and analysis of concerns.

#### 4.4 ROUNDS OF ACTION RESEARCH

We designed our study to evolve through rounds of interaction with the raters. With each round, we took the opportunity to enhance our method and tool according to raters’ opinions and suggestions. Through the next sections (1 section per round, in a total of 3 rounds), we discuss how we materialized the two last phases of the reporting template proposed by Santos and Travassos (2011):

1. Evaluation and Analysis: it must describe the data analysis process and its findings. Santos and Travassos (2011) emphasizes that the data analysis must keep an explicit link between the results obtained from the collected data (the tagged dataset from raters, in our case) and the study’s goal(s). This preserves the traceability of the outcomes to the diagnosed problem and assists in giving rigor to the research;
2. Reflections and Learning: it is responsible for exploring the results of the study in comparison to the state of art. Additionally, study’s results must depict the learning experience of the participants. We highlight that we also learned as researchers as we gradually became aware of the problems and suggestions reported by the specialists through the rounds of evaluation. As stated by Staron (2020), through

---

<sup>4</sup><http://wiki.dcc.ufba.br/PGComp/>

co-development, researchers and practitioners learn from each other, and thus they develop findings which contribute to both the practice and academic theories, tools, methods, and knowledge development.

Action research studies allow researchers to return to the **Diagnostic** phase after going through “Evaluation and Analysis” and “Reflections and Learning”. According to Santos and Travassos (2011) this is not mandatory and researchers may decide to return to the template’s early phases only if they think it is necessary, as in face of circumstance that require redesigning the study. As an example, they mention the following situation: “the researcher can return to the diagnosis stage after an initial attempt to plan the study has not been completed because of a lack of better problem description”. Except for lowering the number of records of the agreement spreadsheets (more information in Section 4.4.1.2), we did not need to modify our study’s initial settings drastically. Therefore, through the next sub-sections, we describe a sequence of evaluation/learning cycles without revisiting **Diagnostic**, **Planning** and **Actions** stages between rounds.

Lastly, as shown in Figure 4.1, we conduct an extra round of investigation by conducting an semi-structured interview with the raters. The following contextualizes the use of semi-structured interviews (LONGHURST, 2003):

*Structured interviews follow a predetermined and standardised list of questions. The questions are always asked in almost the same way and in the same order. At the other end of the continuum are unstructured forms of interviewing such as oral histories... The conversation in these interviews is actually directed by the informant rather than by the set questions. In the middle of this continuum are semi-structured interviews. This form of interviewing has some degree of predetermined order but still ensures flexibility in the way issues are addressed by the informant*

We opted for semi-structured interviews because, although the interviewer prepares a list of predetermined questions, semi-structured interviews are also conversational and they offer to participants the chance to explore issues they feel are important (LONGHURST, 2003). We believe that this aspect has potential to gather more knowledge from the raters as it allows them to elaborate more about their considerations and provide suggestions regarding our method. We also take the opportunity to elucidate some controversial questions that emerged during the rounds of evaluation.

#### 4.4.1 Round 1

Running our study’s rounds required sending evaluation packages to raters. The packages exposed some information concerning our rationale to take advantage of raters’ know-how. For instance, they received a copy of one of our papers (CARVALHO; NOVAIS; MENDONÇA, 2018). The paper contains some details about how our method works. We wanted the specialists to grow more familiarized with our method and tool, so that, with each round of evaluation, they could provide us with deeper insights and suggestions. We also sent to them: (i) a spreadsheet written in the format exhibited in Table 4.2 and

(ii) systems’ source code files from which we extracted concerns. Explicitly exposing our line of reasoning regarding concerns identification sides with one of the primary goals of action research (SANTOS; TRAVASSOS, 2011): reaching a self-changing process that is not only observed by researchers but it is also influenced by them. Plus, researchers must be able to interpret the field under investigation, plan and conduct interventions, formulate concepts and theories, and prepare explanations. We explored many of these dimensions through the rounds of our study.

#### 4.4.1.1 Evaluation and Analysis

After both raters sent us the tagged datasets, we ran them through the kappa coefficient. Considering a simple percentage calculation, they agreed with 74.2% of the concerns identified by AKS. We reached 0.28 in the kappa’s agreement scale, *i.e.*, a “Fair” strength of agreement, as exhibited in Table 2.2 (in Section 2.8). This means: although the raters agreed with the majority of the identified concerns (74.2%), their opinions did not align enough to grant us a higher level agreement. We then consulted the information they inserted in the *Comment* column of the dataset. In next sections, we provide further details about their comments and how we used them to adapt and improve our method and tool.

#### 4.4.1.2 Reflections and Learning

Skipping the analysis of annotations was one of the first problems that raters identified. Annotations are constructs for declaratively associating additional metadata information to program elements. The extra metadata can be used for different purposes, such as: guidance for the compiler, compile-time or deployment-time processing, and runtime processing (YU et al., 2018). For instance, the following source code fragment (in Listing 4.1) shows one of the classes that the raters analyzed. Developers added test related annotations throughout the source code of the **WebServerTest** class (lines highlighted in green): *@After* (line 27) and *@Test* (*e.g.*, lines 32, 37, 43). According to raters, this class’ DtC should be categorized as high considering the “Test” concern. However, AKS classified it as slight because it did not process the annotations. We found several other examples of annotations being used to implement concerns in the source code of the analyzed projects.

**Listing 4.1** Test Annotations

```

1 package org.kairosdb.core.http;
2
3 import com.google.common.io.Resources;
4 import org.apache.http.conn.HttpHostConnectException;
5 import org.junit.After;
6 import org.junit.Test;
7 import org.kairosdb.core.exception.KairosDBException;
8 import org.kairosdb.testing.Client;
9 import org.kairosdb.testing.JsonResponse;

```

```
10
11 import java.io.IOException;
12 import java.net.UnknownHostException;
13 import java.security.KeyManagementException;
14 import java.security.KeyStoreException;
15 import java.security.NoSuchAlgorithmException;
16 import java.security.UnrecoverableKeyException;
17 import java.security.cert.CertificateException;
18
19 import static org.hamcrest.CoreMatchers.equalTo;
20 import static org.hamcrest.Matchers.greaterThan;
21 import static org.junit.Assert.assertThat;
22
23 public class WebServerTest {
24     private WebServer server;
25     private Client client;
26
27     @After
28     public void tearDown() {
29         if (server != null) server.stop();
30     }
31
32     @Test(expected = NullPointerException.class)
33     public void test_constructorNullWebRootInvalid() throws
34         UnknownHostException {
35         new WebServer(0, null);
36     }
37
38     @Test(expected = NullPointerException.class)
39     public void test_setSSLSettings_nullKeyStorePath_invalid() throws
40         UnknownHostException {
41         server = new WebServer(0, ".");
42         server.setSSLSettings(443, null, "password");
43     }
44
45     @Test(expected = IllegalArgumentException.class)
46     public void test_setSSLSettings_emptyKeyStorePath_invalid() throws
47         UnknownHostException {
48         server = new WebServer(0, ".");
49         server.setSSLSettings(443, "", "password");
50     }
51
52     @Test(expected = NullPointerException.class)
53     public void test_setSSLSettings_nullKeyStorePassword_invalid()
54         throws UnknownHostException {
55         server = new WebServer(0, ".");
56         server.setSSLSettings(443, "path", null);
57     }
58
59     @Test(expected = IllegalArgumentException.class)
```

```

56  public void test_setSSLSettings_emptyKeyStorePassword_invalid ()
      throws UnknownHostException {
57      server = new WebServer(0, ".");
58      server.setSSLSettings(443, "path", "");
59  }
60
61  @Test
62  public void test_setSSLCipherSuites_emptyCipherSuites_valid ()
      throws UnknownHostException {
63      server = new WebServer(0, ".");
64      server.setSSLCipherSuites("");
65  }
66
67  @Test(expected = NullPointerException.class)
68  public void test_setSSLCipherSuites_nullCipherSuites_invalid ()
      throws UnknownHostException {
69      server = new WebServer(0, ".");
70      server.setSSLCipherSuites(null);
71  }
72
73  @Test
74  public void test_setSSLProtocols_emptyProtocols_valid () throws
      UnknownHostException {
75      server = new WebServer(0, ".");
76      server.setSSLProtocols("");
77  }
78
79  @Test(expected = NullPointerException.class)
80  public void test_setSSLProtocols_nullProcotol_invalid () throws
      UnknownHostException {
81      server = new WebServer(0, ".");
82      server.setSSLProtocols(null);
83  }
84
85  @Test
86  public void test_constructorNullAddressValid () throws
      UnknownHostException {
87      WebServer webServer = new WebServer(null, 0, ".");
88
89      assertEquals("localhost", webServer.getAddress().getHostName());
90  }
91
92  @Test
93  public void test_constructorEmptyAddressValid () throws
      UnknownHostException {
94      WebServer webServer = new WebServer("", 0, ".");
95
96      assertEquals("localhost", webServer.getAddress().getHostName());
97  }

```

```

98
99  @Test
100  public void test_SSL_success()
101      throws KairosDBException, IOException,
102             UnrecoverableKeyException, CertificateException,
103             NoSuchAlgorithmException, KeyStoreException,
104             KeyManagementException,
105             InterruptedException {
106      String keyStorePath = Resources.getResource("keystore.jks").
107          getPath();
108      String keyStorePassword = "testing";
109      server = new WebServer(0, ".");
110      server.setSSLSettings(8443, keyStorePath, keyStorePassword);
111      ...
112  }
113
114  @Test(expected = HttpHostConnectException.class)
115  public void test_noSSL()
116      throws KairosDBException, IOException,
117             UnrecoverableKeyException, CertificateException,
118             NoSuchAlgorithmException, KeyStoreException,
119             KeyManagementException,
120             InterruptedException {
121      String keyStorePath = Resources.getResource("keystore.jks").
122          getPath();
123      String keyStorePassword = "testing";
124      server = new WebServer(0, ".");
125      ...
126  }
127
128  @Test
129  public void test_SSL_and_HTTP_success()
130      throws KairosDBException, IOException,
131             UnrecoverableKeyException, CertificateException,
132             NoSuchAlgorithmException, KeyStoreException,
133             KeyManagementException,
134             InterruptedException {
135      String keyStorePath = Resources.getResource("keystore.jks").
136          getPath();
137      String keyStorePassword = "testing";
138      server = new WebServer(9001, ".");
139      server.setSSLSettings(8443, keyStorePath, keyStorePassword);
140      ...
141  }
142
143  @Test
144  public void test_basicAuth_unauthorized()
145      throws KairosDBException, IOException, InterruptedException {
146      server = new WebServer(9001, ".");
147      server.setAuthCredentials("bob", "bobPassword");
148      ...
149  }

```

```

141
142  @Test
143  public void test_basicAuth_authorized ()
144      throws KairosDBException , IOException , InterruptedException {
145      server = new WebServer(9001, ".");
146      server.setAuthCredentials("bob", "bobPassword");
147      ...
148  }
149  }

```

Even though we reduced the agreement dataset down to 230 records, raters complained about the excess of effort required by the manual verification of source code samples. Consequently, we decided to reduce the dataset even more after a brief discussion with the raters. Basically, we asked them what number of records they thought would be enough to tag comfortably. We ended up with 120 records. Such a smaller amount of records did not jeopardize the precision of our evaluation as we noticed that there are not so many possibilities about how to analyze the relationship between source code artifacts and concerns. As a consequence, our initial 230-records-long agreement dataset contained several repeated cases of association between artifacts and concerns. We concluded that a 120 records-long dataset would suffice for evaluating our approach while avoiding fatiguing the raters.

As another way to help the raters to analyze source code samples, we took the opportunity to expand ASK' concerns manipulation capabilities. To help raters to visually identify the implementation of concerns, we added annotating routines to AKS. We made it possible for AKS to add annotations to locate pieces of the source code that implement a concern. For example, in Listing 4.2, the import of “`javax.validation.Valid`” is associated with the “Validation” concern. AKS adds the `@Concern` annotation to indicate the elements of the source code (highlighted in green) that are used to implement this concern (lines 5, 13, and 19).

**Listing 4.2** @Concern Annotation

```

1  package org.kairosdb.core.http.rest.json;
2
3  import org.codehaus.jackson.annotate.JsonCreator;
4
5  @Concern(name="Validation")
6  import javax.validation.Valid;
7  import java.util.Collections;
8  import java.util.List;
9
10 public class MetricRequestList {
11     @Valid List<NewMetricRequest> metricsRequest;
12
13     @Concern(name = "Validation")
14     @JsonCreator
15     public MetricRequestList (List<NewMetricRequest> metricsRequest) {
16         this.metricsRequest = metricsRequest;
17     }

```



```

18
19  @Concern(name = "Validation")
20  public List<NewMetricRequest> getMetricsRequest() {
21      return Collections.unmodifiableList(metricsRequest);
22  }
23 }

```

The addition of concerns-related annotations might help us to expand the use of our method in the future. It can be used by developers to indicate concerns that AKS cannot detect. For instance, the annotations can enable the mapping of concerns that are implemented without the injection of third-party components.

Arguably, embedding annotations in the source could persuade the raters to focus only on the annotated parts during the tagging. Being aware of this threat, we instructed them to consider the annotations merely as indicators of the items that AKS processes to spot concerns and their opinions should always prevail above this. In other words, we stressed to raters that they could use the annotations to track the elements which AKS analyzes while it evaluates the association between source code artifacts and concerns. They should keep on criticizing our categorization of concerns according to their own point of view.

One of the raters did not agree with categorizing java *interfaces*' DtC. This means: he/she agreed about the fact that the *interfaces* were actually linked to some concerns through imported components, but he/she could not determine the value of DtC without the methods' bodies of code. We had two options to manage this situation: (i) perform the evaluation only on artifacts whose classes have methods with body of code (and exclude *interfaces*); or (ii) try to persuade the rater to evaluate the DtC of *interfaces*. We opted for the second and confronted him/her with the following rationale:

*Although the lack of methods' body of code can make the categorization of DtC difficult, there is still a possibility if the interface's methods' declarations are taken into account. For instance, someone can categorize DtC by considering the types of parameters and return declarations. The resulting relationship between the interface and such types reflects the actual desire of developers to determine which concerns the interface's children classes must implement. In this moment, a relationship is conceptually established between the interface and the concerns.*

#### 4.4.2 Round 2

We carried out the aforementioned improvements and adaptations. We also asked the rater who refused to analyze interfaces to reconsider his/her opinion. We randomly (re)selected new source code samples and generated a new agreement spreadsheet. This means: the source code snippets that raters analyzed in the second round differed from those that we sent to them in the first round. We checked the new samples to assure that they contained examples related to the problems that the raters identified in the first round before sending the new evaluation package. Doing this is important because we wanted them to re-evaluate the same cases and provide new opinions about them.

#### 4.4.2.1 Evaluation and Analysis

This time, the processing of kappa gave us 0.26 (strength of agreement = “Fair”). They agreed with 83.2% of the concerns that AKS identified and categorized according to our DtC metric. As we were not satisfied with the strength of agreement, once again, we counted on the raters’ comments to refine our method and AKS.

#### 4.4.2.2 Reflections and Learning

Unfortunately, the rater who did not want to evaluate *interfaces* did not agree with our argument. His/her decision impacted our action research study. We decided to: (i) not remove the processing of *interfaces* from our method and tool. As the other rater did not complain about tagging *interfaces*, we want to make our method available to anyone interested in analyzing this type of class<sup>5</sup>; (ii) exclude *interfaces* from the agreement dataset. The raters would not agree about the categorization of concerns because one of them would never give his/her opinion about *interfaces*.

Our method classified the source code exhibited in Listing 4.3 as moderately dedicated to implement the “Database” concern (a moderate DtC), but one the raters did not agree. He/She pointed out that DtC should be categorized as high because all non-empty methods of the class focus on the implementation of the concern. Then, we modified AKS to skip processing empty methods like `NoopSuggestBackendReporter` and `reportWriteDroppedByRateLimit` (lines 9 and 18, in gray). Actually, the inclusion of empty methods dilutes DtC, as the MD metric calculates a ratio between the number of methods that reference a concern (NOR) and the total number of artifacts’ methods (NOM). As the rater pointed out that it is not possible to determine the DtC of methods that have no body of code, they should not take part in the measurement.

**Listing 4.3** Moderate DtC from empty methods

```

1 package com.spotify.heroic.statistics.noop;
2
3 @Concern(name="Database")
4 import com.spotify.heroic.statistics.SuggestBackendReporter;
5 @Concern(name="Database")
6 import com.spotify.heroic.suggest.SuggestBackend;
7
8 public class NoopSuggestBackendReporter implements
9     SuggestBackendReporter {
10     private NoopSuggestBackendReporter() {}
11     @Concern(name = "Database")
12     @Override
13     public SuggestBackend decorate(final SuggestBackend backend) {
14         return backend;
15     }
16

```

---

<sup>5</sup>The processing or exclusion of *interfaces* can be parameterized during the execution of AKS.

```

17  @Override
18  public void reportWriteDroppedByRateLimit() {}
19
20  private static final NoopSuggestBackendReporter instance = new
    NoopSuggestBackendReporter ();
21
22  @Concern(name = "Database")
23  public static NoopSuggestBackendReporter get() {
24      return instance;
25  }
26 }

```

The raters mentioned cases in which our method failed to associate components with concerns. The source code excerpt in Listing 4.4 illustrates this situation. From line 16 to 29, **OpenCensusApplicationEventListener** class imports several components (highlighted in yellow) to implement the “Web App Support” concern (in green). However, one of the raters argued that the “`javax.ws.rs.*`” (lines 14 and 15, in yellow) should also be associated with this concern. In other words, we associated all components identified as “`javax.ws.rs.*`” with the “Service-Orientation” concern because they are often used to implement restful web services. The rater noticed that the DtC would be different if we also associated “`javax.ws.rs.*`” with the “Web App Support” concern. According to his/her opinion: implementing restful web services is a way to add “web app support” to a system. We accepted this observation and modified our method and AKS to allow the overlapping of associations between concerns and imported components, *e.g.*, associating both “Service-Orientation” and “Web App Support” with the import of the “`javax.ws.rs.*`” component.

**Listing 4.4** Overlapping of Concerns

```

1  package com.spotify.heroic.http.tracing;
2
3  import static java.text.MessageFormat.format;
4
5  import io.opencensus.trace.AttributeValue;
6  import io.opencensus.trace.Span;
7  import io.opencensus.trace.Status;
8  import io.opencensus.trace.Tracer;
9  import io.opencensus.trace.Tracing;
10 import java.util.ArrayList;
11 import java.util.List;
12 import java.util.Map;
13 import java.util.stream.Collectors;
14 import javax.ws.rs.container.ContainerRequestFilter;
15 import javax.ws.rs.container.ContainerResponseFilter;
16 @Concern(name="Web App Support")
17 import org.glassfish.jersey.server.ContainerRequest;
18 @Concern(name="Web App Support")
19 import org.glassfish.jersey.server.ContainerResponse;
20 @Concern(name="Web App Support")

```

```

21 import org.glassfish.jersey.server.model.Invocable;
22 @Concern(name="Web App Support")
23 import org.glassfish.jersey.server.monitoring.ApplicationEvent;
24 @Concern(name="Web App Support")
25 import org.glassfish.jersey.server.monitoring.ApplicationEventListener;
26 @Concern(name="Web App Support")
27 import org.glassfish.jersey.server.monitoring.RequestEvent;
28 @Concern(name="Web App Support")
29 import org.glassfish.jersey.server.monitoring.RequestEventListener;
30
31 class OpenCensusApplicationEventListener implements
    ApplicationEventListener {
32     private final Tracer globalTracer = Tracing.getTracer();
33     private final OpenCensusFeature.Verbosity verbosity;
34
35     ...

```

The following are other concerns whose categories we overlapped following the same suggestion: (i) “Encryption” and “Security”: as encryption is often used to implement security; (ii) “Tracing” and “Monitoring”: a tracing operation can serve as a way to monitor a system; and (iii) “Graph Computing” and “Mathematical Processing”: we noticed that routines used to support graph computing are often based on mathematical processing of data. So, we added special structures and routines to AKS to support the overlapping of concerns.

One of the raters warned us that we were missing some concerns. He/she noticed that we were not parsing all POM/Gradle files of the projects because developers may scatter the information about third-party components. Figure 4.3 exemplifies this situation. Developers of Heroic embedded many POM files in its source code. First versions of AKS was capable of parsing only the main POM file stored in the root folder of projects, which is the usual location of POMs (highlighted in red). We then enabled AKS to mine concerns from several scattered POM/Gradle files (in blue).

### 4.4.3 Round 3

We restarted the study after implementing the aforementioned modifications and additions. We sent a new agreement dataset to raters after selecting new random samples of code snippets. Unfortunately, one of the raters stopped replying to our requests to participate in the third round of evaluation. So, we had to replace him/her.

#### 4.4.3.1 Evaluation and Analysis

We obtained 0.55 from kappa (strength of agreement = “Moderate”) after the raters tagged the dataset. Considering a simple percentage calculation, they agreed with 84.2% of AKS’ diagnoses. We then decided to end the study. We discuss more details about this decision through the next sub-sections.

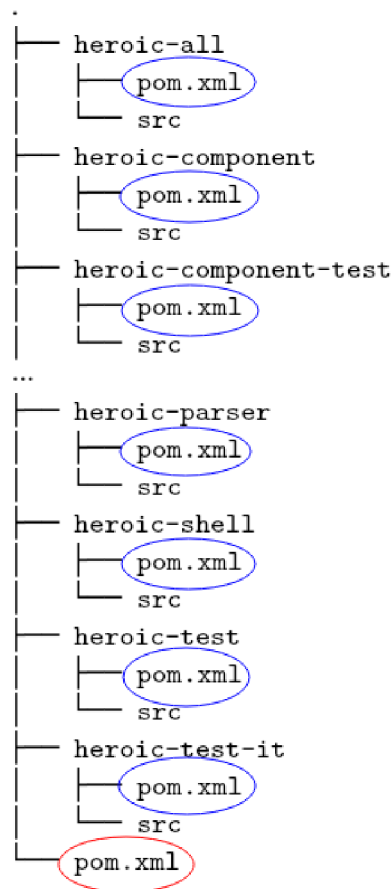


Figure 4.3 Heroic’s Tree of POM Files

#### 4.4.3.2 Reflections and Learning

It is debatable that replacing one of the raters could break the succession of contributions to our study as, with each round, raters grew more knowledgeable about our method and could keep on raising insightful questions and suggestions. However, we replaced only one of them while keeping the other, and we assured that the new rater had the necessary academic and professional experience to help us conducting our study.

Our method applies static analysis on source code artifacts to find concerns and categorize the DtC between them. Thus, it is restrained to what we can achieve from the Abstract Syntax Trees (AST) extracted from the artifacts. As a consequence, our method may lack a precision that semantically-enriched approaches can accomplish. For instance, we categorized as slightly dedicated the relationship between the source code excerpt exhibited in Listing 4.5 and the “Geospatial Processing” concern (lines 6, 8, 47, and 68, in green). One of the raters disagreed because he took other aspects into account: many occurrences of the word “spatial” (from line 17 to 22, in gray) suggests that the artifact focus on demonstrating how a spatial search is performed. AKS did not process these pieces of information because MVNRepository categorize “lucene” as a

“Full-Text Indexing Library”<sup>6</sup> and, accordingly, we chose the “Text Processing” concern to represent it. If we had enabled AKS to process keywords and associate them with concerns via semantic processing mechanisms, we would have an additional resource to refine the identification of concerns: associating the “spatial” keyword of lucene import declarations with “geospatial processing”.

**Listing 4.5** Using Parts of Other Imports to Refine DtC

```

1 package org.janusgraph.diskstorage.lucene;
2
3 import com.google.common.base.Preconditions;
4 import com.google.common.collect.ImmutableMap;
5 import com.google.common.collect.Sets;
6 @Concern(name="Geospatial Processing")
7 import org.locationtech.spatial4j.context.SpatialContext;
8 @Concern(name="Geospatial Processing")
9 import org.locationtech.spatial4j.shape.Shape;
10 import org.janusgraph.core.attribute.Geoshape;
11 import org.janusgraph.util.system.IOUtils;
12 import org.apache.lucene.analysis.Analyzer;
13 import org.apache.lucene.analysis.standard.StandardAnalyzer;
14 import org.apache.lucene.document.*;
15 import org.apache.lucene.index.*;
16 import org.apache.lucene.search.*;
17 import org.apache.lucene.spatial.SpatialStrategy;
18 import org.apache.lucene.spatial.prefix.RecursivePrefixTreeStrategy;
19 import org.apache.lucene.spatial.prefix.tree.GeohashPrefixTree;
20 import org.apache.lucene.spatial.prefix.tree.SpatialPrefixTree;
21 import org.apache.lucene.spatial.query.SpatialArgs;
22 import org.apache.lucene.spatial.query.SpatialOperation;
23 import org.apache.lucene.store.Directory;
24 import org.apache.lucene.store.FSDirectory;
25 import org.junit.Before;
26 import org.junit.Test;
27
28 ...
29
30 /** @author Matthias Broecheler (me@matthiasb.com) */
31 public abstract class LuceneExample {
32
33     public static final File path = new File("/tmp/lucene");
34     private static final String STR_SUFFIX = "_str";
35     private static final String TXT_SUFFIX = "_txt";
36
37     private static final int MAX_RESULT = 10000;
38
39     private final Map<String, SpatialStrategy> spatial = new HashMap

```

<sup>6</sup>Apache describes it as a “full-featured text search engine library written entirely in Java” (<https://lucene.apache.org/core/>)

```

    <>());
40     private final SpatialContext ctx = SpatialContext.GEO;
41
42     @Before
43     public void setup() {
44         ...
45     }
46
47     @Concern(name = "Geospatial Processing")
48     private SpatialStrategy getSpatialStrategy(String key) {
49         SpatialStrategy strategy = spatial.get(key);
50         if (strategy == null) {
51             final int maxLevels = 11;
52             SpatialPrefixTree grid = new GeohashPrefixTree(ctx, maxLevels)
53                 ;
54             strategy = new RecursivePrefixTreeStrategy(grid, key);
55             spatial.put(key, strategy);
56         }
57         return strategy;
58     }
59     @Test
60     public void example1() throws Exception {
61         ...
62     }
63
64     private Set<String> getResults(IndexSearcher searcher, TopDocs
65         docs) throws IOException {
66         ...
67     }
68     @Concern(name = "Geospatial Processing")
69     void indexDocs(IndexWriter writer, String documentId, Map<String,
70         Object> docMap)
71         throws IOException {
72         Document doc = new Document();
73         Field documentIdField = new StringField("docid", documentId,
74             Field.Store.YES);
75         doc.add(documentIdField);
76         for (Map.Entry<String, Object> kv : docMap.entrySet()) {
77             String key = kv.getKey();
78             Object value = kv.getValue();
79
80             if (value instanceof Number) {
81                 final Field field;
82                 if (value instanceof Integer || value instanceof Long) {
83                     field = new LongPoint(key, ((Number) value).longValue());
84                 } else { // double or float
85                     field = new DoublePoint(key, ((Number) value).doubleValue
86                         ());

```

```

86     }
87     doc.add(field);
88     } else if (value instanceof String) {
89     String str = (String) value;
90     Field field = new TextField(key + TXT.SUFFIX, str, Field.
        Store.NO);
91     doc.add(field);
92     if (str.length() < 256) field = new StringField(key +
        STR.SUFFIX, str, Field.Store.NO);
93     doc.add(field);
94     } else if (value instanceof Geoshape) {
95     Shape shape = ((Geoshape) value).getShape();
96     for (IndexableField f : getSpatialStrategy(key).
        createIndexableFields(shape)) {
97     doc.add(f);
98     }
99     } else throw new IllegalArgumentException("Unsupported type: "
        + value);
100 }
101
102 writer.updateDocument(new Term("docid", documentId), doc);
103 }
104 }

```

AKS determined that the source code in Listing 4.6 is highly dedicated to implement the “Validation” concern (in green, in lines 3, 8, 15, 24, 30, 51, and 61) because of the use of components responsible for performing validations (lines 4, 9, and 16, in yellow). There is actually a strong relationship between the artifact and “Validation” throughout lines of code, but we could refine our method and AKS if we considered some of its semantic elements. For instance, the description provided in line 18 (in gray). It is a clear statement about the purpose of the class: “groups data points by tag names”. The statement convinced one of the raters to disagree with the categorization of DtC as high because it points to a purpose other than “Validation”. He/she categorized it as moderate.

**Listing 4.6** Using Descriptions to Refine DtC

```

1 package org.kairosdb.core.groupby;
2
3 @Concern(name="Validation")
4 import org.apache.bval.constraints.NotEmpty;
5 import org.kairosdb.core.DataPoint;
6 import org.kairosdb.core.aggregator.annotation.GroupByName;
7
8 @Concern(name="Validation")
9 import javax.validation.constraints.NotNull;
10 import java.util.ArrayList;
11 import java.util.Collections;
12 import java.util.List;
13 import java.util.Map;
14

```



```

15  @Concern(name="Validation")
16  import static com.google.common.base.Preconditions.checkNotNull;
17
18  @GroupName(name = "tag", description = "Groups data points by tag names")
19  public class TagGroupBy implements GroupBy {
20      @NotNull @NotEmpty private List<String> tags;
21
22      public TagGroupBy() {}
23
24      @Concern(name = "Validation")
25      public TagGroupBy(List<String> tagNames) {
26          checkNotNull(tagNames);
27          this.tags = new ArrayList<String>(tagNames);
28      }
29
30      @Concern(name = "Validation")
31      public TagGroupBy(String... tagNames) {
32          this.tags = new ArrayList<String>();
33          Collections.addAll(this.tags, tagNames);
34      }
35
36      @Override
37      public int getGroupId(DataPoint dataPoint, Map<String, String>
38          tags) {
39          // Never used. Grouping by tags are done differently for
40          // performance reasons.
41          return 0;
42      }
43
44      @Override
45      public GroupByResult getGroupByResult(int id) {
46          // Never used. Grouping by tags are done differently for
47          // performance reasons.
48          return null;
49      }
50
51      @Override
52      public void setStartDate(long startDate) {}
53
54      @Concern(name = "Validation")
55      /**
56       * Returns the list of tag names to group by.
57       *
58       * @return list of tag names to group by
59       */
60      public List<String> getTagNames() {
61          return Collections.unmodifiableList(tags);
62      }
63
64      @Concern(name = "Validation")

```

```

62     public void setTags(List<String> tags) {
63         this.tags = tags;
64     }
65 }

```

We determined that the code snippet in Listing 4.7 is highly associated with the “Database” concern. AKS processed the imports in lines 6, 8 and 10 (highlighted in yellow) and methods in line 15 and 21, and ended up with a high DtC. One of the raters disagreed and pointed out that developers designed the artifact to perform tests. He/She pointed out that the name of the class (in grey, in line 13) is more related to “Test” than to the “Database” concern. According to him/her, DtC should be slight regarding “Database”.

**Listing 4.7** Using Class Names to Refine DtC

```

1  package org.janusgraph.graphdb.embedded;
2
3  import org.junit.BeforeClass;
4
5  @Concern(name="Database")
6  import org.janusgraph.CassandraStorageSetup;
7  @Concern(name="Database")
8  import org.janusgraph.diskstorage.configuration.WriteConfiguration;
9  @Concern(name="Database")
10 import org.janusgraph.graphdb.JanusGraphPerformanceMemoryTest;
11
12 /** @author Matthias Broecheler (me@matthiasb.com) */
13 public class EmbeddedGraphMemoryPerformanceTest extends
14     JanusGraphPerformanceMemoryTest {
15     @Concern(name = "Database")
16     @BeforeClass
17     public static void startCassandra() {
18         CassandraStorageSetup.startCleanEmbedded();
19     }
20
21     @Concern(name = "Database")
22     @Override
23     public WriteConfiguration getConfiguration() {
24         return CassandraStorageSetup.
25             getEmbeddedCassandraPartitionGraphConfiguration(
26                 getClass().getSimpleName());
27     }

```

The aforementioned examples made us perceive that there is a limit to how the processing of AST can replicate raters’ comprehension about the association between source code and concerns. We believe that refining our method and tool require combining AST-based analysis with others that can take advantage of semantic elements found in

the artifacts: names and keywords extracted from packages, fields, classes, methods and descriptive annotations. We define “semantic elements” as a facet of concerns identification and analysis that is related to the meaning of words (as “Test” in the name of class in Listing 4.7) and sentences (as in the description of the annotation in Listing 4.6).

In Table 4.3 we summarize the evolution of our action research study. The *Round* column identifies each of the three rounds of the study. The second column, *Confirmations*, displays a simple percentage calculation based on the number of positive answers (“YES”) provided by the raters when they agreed with the analysis performed by AKS. The third column shows the evolution of the *Strength of Agreement*. It is important to notice that a high percentage of confirmations does not always indicate an adequate agreement (as in round 1 and 2). The Cohen’s kappa strength of agreement complements and reinforces the percentage-based evaluation by ensuring the alignment of raters’ opinions.

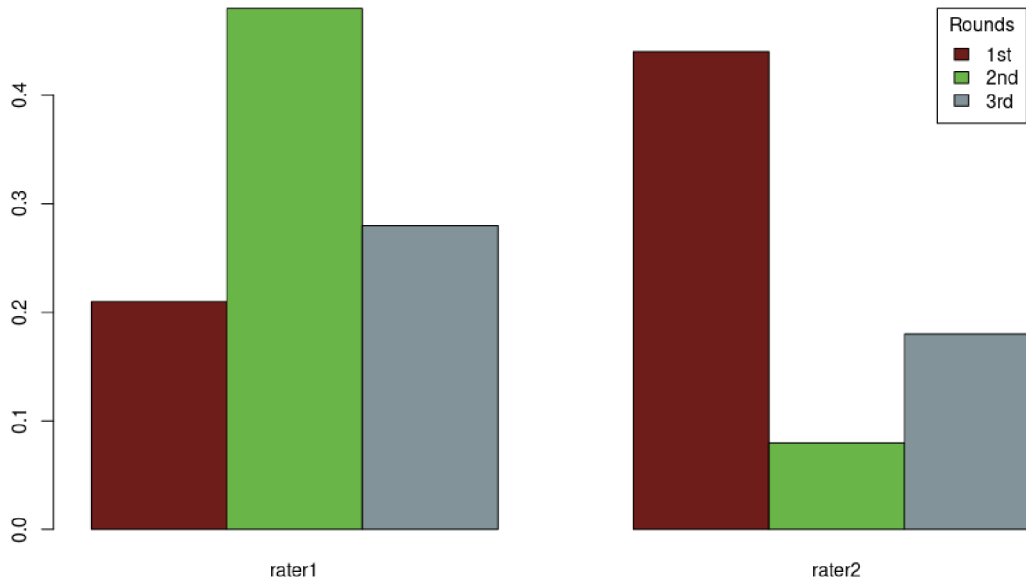
**Table 4.3** Evolution of our Action Research Study

Round	Confirmations %	Strength of Agreement
1	74.2	0.28 (Fair)
2	83.2	0.26 (Fair)
3	84.2	0.55 (Moderate)

Figure 4.4 shows a distribution of raters’ comments per round. The bars’ heights correspond to a simple percentage obtained from the number of comments added to agreement dataset by raters. Rater1 (on the left side of the figure) was responsible for the highest number of comments. This comes partially, for instance, from him/her refusing to analyze the DtC of *interfaces* and methods of *abstract* classes, which we were still adding to the dataset up until the second round. Therefore, he/she would complain (in the comments) about having to tag them. Rater2’s comments peaked in the first round and reduced during the second and third ones. As he/she became satisfied, his/her later opinions (in second and third rounds) grew more dispersed and sporadic. We must point out that we replaced rater1 before running the third round. He/she stopped responding to our invitations to participate in the study. The “new rater1” tended to agree with the third round’s spreadsheet’s data because we had already refined our method and tool to a satisfactory point after the first two rounds.

Table 4.4 summarizes some other few general aspects that we observed from what the raters filled in the study’s spreadsheets.

We decided to end our action research study after its third round because: (i) reaching a strength of agreement beyond 0.40 (strength of agreement = “Moderate”) is considered a good agreement (DONKER; HASMAN; GEIJN, 1993)(MUNOZ; BANGDIWALA, 1997), which gave us enough precision to conduct other studies (*e.g.*, Study II in Chapter 5); and (ii) we must evolve our method to consider the processing of source code’s semantic elements. This is a complex new aspect of our research and it deserves a deeper study with its own scope of goals and investigation approaches.



**Figure 4.4** Comments by Rater and Round

**Table 4.4** Tendencies regarding Raters' Contributions

Aspect	Tendency
Identification of Concerns	They usually agreed with the identification, but we spotted some exceptions
Measurement of DtC	When they disagreed with how we measured the DtC they would preferably point out HIGH (High Dedication) as the correct value
List of imports	Raters did not usually comment about the list of imports (that we used to identify concerns). However, we noticed some few exceptions in raters' responses during round 2
Details in the Comments	One of the raters declined from adding more details to comments after the first round <sup>a</sup>

<sup>a</sup> As we pointed in Figure 4.4 rater2 became satisfied after the first round. This contributed for his/her comments' lack of details later on.

#### 4.4.4 A Semi-structured Interview

After finishing the third round of evaluation, we conducted a semi-structured interview with the raters. We presented the results of the study and took the opportunity to clarify some questions and gather some extra feedback from them. We divided the interview into two parts. In the first part, we presented the results and collected some impressions and opinions. We took manual notes of what the raters said and arranged them in a new

presentation. During the second part of the interview, we re-presented the results and discussed their opinions once again. Splitting the interview gave us a time to think about raters' feedback and to compile the body of knowledge that we acquired from them. This proved to be an advantageous maneuver as it enabled us to better absorb raters' feedback and discuss study's findings in a more systematic manner during the second part of the interview.

One important result that we discussed with the raters was the fact one of them did not want to evaluate the DtC of *interfaces* and methods of *abstract* classes. The rater who refused to evaluate the DtC did not participate in the interview because he/she was the one who stopped responding to our invitations to take part in the third round. The raters agreed that methods' declarations and their list of parameters and result types are good indicators of which concerns are being implemented, but this is subject to the impact that the *interfaces* have on the development of systems. If a particular system relies on the definition of many abstract modules (*e.g.*, *interfaces* and *abstract* classes), it might be insightful to process these types of artifacts. The raters see an advantage in perceiving how concerns are firstly introduced in software projects at an abstract level before their realization (as concrete classes).

One of the raters mentioned examples of concerns that are implemented exclusively with the help of *interfaces*. For instance, Java Persistence API (JPA)<sup>7</sup> is a standard for connecting applications to databases. Some of its extensions<sup>8</sup> relies on annotated *interfaces* and developers rarely need to add extra lines of code. According to him/her, it is important to associate the "Database" concern with JPA-annotated *interfaces*. In this case, the processing of *interfaces* is highly desirable.

The raters agreed that parameterizing our method and AKS to either process or skip *interfaces* and *abstract* methods is a good maneuver. Users of AKS will have how to adapt the tool to their needs.

We revealed to raters that we would like to enable our method and AKS to process semantic aspects of software projects. They both agreed that in many occasions they could not agree with the results listed in the agreement dataset because of the non-processing of artifacts' semantic elements. They also questioned whether the definition of a "glossary" (or a taxonomy, or an ontology) would have to precede the semantic analysis of source code. For example, the "Test" concern may appear in different forms. Not rarely, developers do not follow naming conventions and they may call "Test" as: "Verification"/"Verify", "Evaluation"/"Evaluate", "AssertThat", and other similar expressions. In other words, providing our method and tool with semantic-oriented processing mechanisms depends on knowing the different ways on how developers name concerns through the codebase.

Although the raters recognized that the semantic processing of concerns is valuable, one of them stressed that the static analysis is more important. According to him/her, the source code elements that AKS processes to identify concerns (import declarations, attributes/parameters types, variable declarations) tend to be "constant" assets in software

---

<sup>7</sup><https://www.oracle.com/java/technologies/persistence-jsp.html>

<sup>8</sup>The rater mentioned the Spring Data JPA (<https://spring.io/projects/spring-data-jpa>)

making. They are intrinsic parts of many programming languages and software systems cannot be easily made without them. Moreover, developers may fail to adequately add/describe/comment/annotate source code’s semantic information. So, the processing of software projects’ semantic elements must be seen as a complementary step to refine the static analysis, not the other way around.

We also presented the results of our next study to the raters. In Section 5.4, we discuss their comments about DtC. Additionally, they proposed some practical applications of our method and tool, which we discuss in Section 7.2.

## 4.5 THREATS TO VALIDITY

Now, we discuss the threats to validity of our action research study:

**Construct validity:** this type of threat questions if we succeeded in measuring the attributes that we needed to measure. We based our method on information about third-party components which developers add to POM and Gradle files. When MVNRepository failed to categorize components, we manually filled this information. We consider that manually filling our own opinions on how to categorize third-part can cause imprecision. However, we circumvent this impact after we manually and attentively reviewed our classification of components/concerns.

We reduced the dataset to enable the raters to tag it easier. We wanted to avoid fatiguing them. The reduction can be seen as detrimental to evaluating our method. However, we soon noticed that there were not many variations in the way how developers use third-party components to implement concerns. Consequently, there are not many ways on how AKS can measure their association with source code artifacts and we are sure that the agreement spreadsheets that we sent to raters contain examples of all possible cases.

**Internal validity:** internal validity is the extent to which evidences support a claim about cause-effect relationships. Categorizing DtC as any value within our scale (slight, moderate and high) may have limited raters’ perception. We wonder if inserting other sub-categories would allow a broader variability of raters’ opinions. Splitting the scale into more qualitative values would cause more variability in the agreement dataset after each tagging. However, the raters did not complain about having to use our scale the same way as they complained about other aspects of our study (*e.g.*, fatigue from having to read too many source code samples).

**External validity:** this threat is related to the degree to which our findings can be generalized. They are restrained to the collection of concerns that we extracted from the projects listed in Table 4.1. We examined real software projects which have counted on the contribution of many developers over a (relatively) long history of releases. However, we did not cover software domains other than the one that we adopted while conducting the evaluation: non-relational databases. It is desirable to consider a broader context of domains, applications, and concerns to generalize our conclusions.

We are positive that having only two raters tagging the dataset granted us with some advantages: (i) two raters suffice for achieving a high evaluation precision via kappa, (ii) it helped to reduce our workload as we did not have to deal with several study’s participants

at the same time, and (iii) a more controlled flow of insights and suggestions that we used to improve our method and tool. However, we must admit that we could have designed our action-research study to include a greater number of specialists to help us generalize our findings even more. Among other effects, it would have forced us to go through more cycles of interactions. In this case, applying Cohen's Kappa to deal with multiple (more than two) raters (CONGER, 1980)(BERRY; JR, 1988) can help replicating this study to solve this threat.

**Conclusion validity:** conclusion validity comprises reasons why conclusions based on an analysis may be incorrect. Although the raters who participated in our study have expertise in software development, they did not have any previous contact with the source code samples that we sent to them. The imprecision resides in raters failing to understand how the samples implement concerns. This is something that only the original developers can do with more accuracy. Therefore, extra studies must refine our dataset and findings with the help of actual systems' development teams. Such deficiency does not discredit our action research's achievements though. We trust that the raters' recommendations and suggestions are beneficial to the application of our method and to future studies.

## 4.6 DISCUSSION

We recommend action research studies as a way to avoid misdirection and promote out-of-box thinking as it enables researchers to count on different viewpoints. As in the following quote (BAETJER, 1997)(PRESSMAN, 2005):

*Software, like all capital, is embodied knowledge and because that knowledge is initially dispersed, tacit, latent, and incomplete in large measure, software development is a social learning process. The process is a dialogue in which the knowledge that must become the software is brought together and embodied in the software. The process provides interaction between users and designers, between users and evolving tools, and between designers and evolving tools [technology]. It is an iterative process in which the evolving tool itself serves as the medium for communication, with each new round of the dialogue eliciting more useful knowledge from the people involved.*

AKS is the evolving tool whose output (the agreement dataset) we used as medium for communicating with stakeholders (the raters). The usefulness of the knowledge which we acquired from interacting with them empowered our method and AKS with ways to: (i) extract consistent concern-related information from the use of third-party components; and (ii) determine the degree of the association between concerns and source code artifacts (DtC).

It took us three rounds of evaluation to reach a good strength of agreement. As informed in Table 4.3 (Section 4.4.3), we achieved a moderate agreement (0.55) and 84% of confirmations after the third round. In other words, apart from reaching a satisfactory agreement, a simple percentage calculation showed that raters agreed that AKS correctly identified the concerns and measured the DtC of 84% of the source code artifacts.

Table 4.5 summarizes raters' opinions and how they impacted our action research

study. We believe that our findings can guide other future studies in determining requirements for dealing with the association between concerns and source code artifacts.

**Table 4.5** Impact of Raters’ Opinions and Reactions

<b>Opinion/Reaction</b>	<b>Rationale</b>	<b>Impact</b>
Inclusion of annotations	Annotated code is also an indication of associations between concerns and source code	Parsing of annotations found in the source code
Fatigue	As performing the analysis tend to produce a large dataset, verification becomes tiresome if done manually	Addition of annotations (@concern) as visual clues to locate concerns and reducing the size of the agreement dataset
Processing of multiple POM and Gradle files	Some developers split the information about components in sub-projects or modules	Identification and processing of scattered POM/Gradle files
Analysis of <i>interfaces</i>	<i>Interfaces</i> lack methods’ body of code. This hinders the identification/verification of concerns and measurement of DtC	Filtering of <i>interfaces</i> out of the dataset when they are not regarded as valuable for concern analysis
Analysis of empty methods	Lacking methods’ body of code disables the identification/verification of concerns and measurement of DtC	Skipping the processing of empty methods of classes
Overlapping of concerns	Some components can refer to more than just one concern	Enabling the configuration of overlapped concerns
Processing of source code’s semantic elements	Reaching higher strengths of agreement may require combining the processing of AST with semantic aspects of the source code	Running future studies to advance our method and tool into the processing of artifacts’ semantic elements

Fatiguing the raters seems a natural consequence of investigating concerns because the manual identification of concerns in large datasets/databases can be tiresome. Therefore, we consider that developers should be helped whenever they need to perform this kind of activity. We did our best to automate the classification of concerns with the help of MVNRepository, as described in Section 3.1.3, but our method still lacks some capabilities



to prevent this task from depending on some manual intervention.

Not all concerns which developers implement depend on the injection of components. Thus, our method can grant only a partial view of the collection of concerns that systems implement. On the other hand, it enables a consistent analysis of all the concerns that can be mined from POM/Gradle files' metadata. This comes from the fact that the information these files enclose represents developers' decisions when they need to implement concerns. In a few words, our method cannot track all concerns that an application contain, but it can guarantee certainty regarding the ones which are extracted from POM/Gradle files.

We did not reveal to the raters the rule and the metrics we used to measure DtC. We wanted them to freely tag the agreement dataset. A variation of our study can do the opposite and present the rule and the metrics with the intention to evaluate them. This may uncover findings that can differ from the ones that we presented in this chapter. By proposing improvements to the rule and metrics, the specialists would have a chance to directly influence the way on how our method processes concerns.

Some of the comments that the raters filled in the agreement spreadsheets are not particular to a specific round of evaluation. For instance, the non-processing of semantic aspects that we discussed in the third round (Section 4.4.3) was already being notified by raters since the first rounds (Sections 4.4.1 and 4.4.2). However, during rounds 1 and 2, raters' suggestions about the processing of static elements of the source code overwhelmed us. After we adapted our method to address such aspects, we began to perceive the semantic facets of concerns mining and analysis.

We decided to end the study after its third round. From our interactions with the raters, we learned that static analysis can replicate the way how they identify concerns to a moderate degree. Therefore, we believe that we can improve our method by joining the processing of AST and semantic elements together. For instance, messages of commits that have historically affected artifacts may encapsulate useful information about concerns. The following is a message found in one of the commits that once updated the source code exhibited in Listing 4.5 (we highlighted some relevant keywords in yellow):

*Add support for indexing line and **polygon geometries**, querying by **polygon geometries**, and support for **geoIntersect**, **geoContains** and **geoDisjoint** predicates. Support for indexing **multi-point/line/polygon** properties is implemented but untested.*

*Signed-off-by: sjudeng sjudeng@users.noreply.github.com*

The message supports the rater's decision to strongly associate the artifact's class, "Luce-Example", with the "Geospatial Processing" concern. Thus, enabling our method to mine and analyze semantic elements of commits' messages is another good approach.

## 4.7 STUDIES THAT USED A SIMILAR APPROACH TO VALIDATE THEIR METHODS

We recommend the work of Santos and Travassos (2011) to understand the use of action research methods. We also highlight a paper published by Staron (2020) who delved into the utilization of action research methods as a potential way to unite computer science researchers and practitioners from the software industry.

As we trust that the work of Santos and Travassos (2011) and Staron (2020) are good sources of information about action research studies, here we focus on investigations that have explored the use of kappa. We want to know more about the application of agreement coefficients in computer science. Specifically, we are interested in the different contexts in which researchers conducted tagging of agreement datasets to evaluate methods and tools: how did they take advantage of raters' expertise and opinions? How many times did the raters have to tag a dataset before reaching a relevant strength of agreement?

Mockus and Votta (2000) looked forward to classify reasons for changes from textual descriptions informed by developers. They compared the classification performed automatically by their tool with the manual classification made by a human agent. Both should fit each change in one distinct category: corrective, adaptive, perfective, and inspection. They reached a high strength of agreement after running the study only once.

Oza and Hall (2005) used kappa to evaluate transcriptions from interviews with software specialists. The interviews had the purpose of eliciting the difficulties in managing offshore software outsourcing. The researchers wanted to extract the major emergent themes from a dataset of keywords. So they conducted an agreement evaluation to avoid the bias of having only one rater to categorize the dataset. They ran the evaluation twice before reaching a significant strength of agreement. In other words, they had to review their classification of themes according to rater's opinions at least once.

While exploring different development processes for creating games, O'Hagan, Coleman, and O'Connor (2014) conducted a study selection pilot to either exclude or include papers in a systematic literature review. Two raters, a researcher and a supervisor, filled their opinions in a dataset about included and excluded studies. It took them only one single interaction to reach a reliable consensus through kappa.

Ma *et al.* (2016) evaluated if the relationship between code smells and fault prediction could be applied to improve the quality of applications. They generated a dataset with information about different types of smells (*e.g.*, blob classes, complex classes, lazy classes, anti-singleton) and conducted an action research study with specialists to determine which smells are more likely to be associated with faults. The study revealed that, in general, smells have little influence on faults. They had to re-run the study because they chose four different software projects to take part in the study, *i.e.*, they ran the study once for each project.

Nayebi *et al.* (2018) tagged a collection of commits as either trivial or non-trivial. They wanted to investigate the impact of commits related to the deletion of source code fragments from mobile applications. They had to run their study only once to reach a significant strength of agreement.

Saikia and Singh (2018) studied a new way to reduce the ambiguity of requirements

elicitation: the use of semantic information contained in requirement documents. Their method benefits from kappa to classify requirements descriptions as either ambiguous or unambiguous. Two annotators (or raters) tagged descriptions obtained from real documents and they reached a considerable strength of agreement after running their agreement study for the first time.

In general, applying an agreement coefficient is useful when it is necessary to rate the categorization of a collection of data while avoiding the bias of having one single specialist to execute the task. Only one of the aforementioned papers ran a study more than once to achieve an agreement. So, we believe that our decision to reboot evaluations in response to raters' opinions is exceptional. We do not see this as a limitation because going through consecutive cycles of evaluations and interactions is one of the expected consequences of conducting action research studies (STARON, 2020).

## 4.8 CONCLUSION

We wanted to validate our method and determine whether it can identify and analyze concerns from third-party components' metadata. Additionally, we needed to reduce the bias of our guesswork regarding the metrics and the rule which we defined to measure DtC.

We are positive that our decision to combine action research with the kappa granted our research the necessary rigor while guaranteeing a certain degree of flexibility. The flexibility resided in going through different rounds of evaluation and having a chance to adapt our method and tool after each round. Kappa helped us to achieve rigor as it prevented us from stopping the evaluation after reaching a high (simple) percentage of approval from raters at the end of the first two rounds.

Conducting a semi-structured interview after the final round of evaluation proved to be very fruitful. We were able to discuss some controversial opinions that we collected from raters. For instance, refusal of one of them to evaluate concerns extracted from *interfaces* and empty methods of *abstract* classes. The interview also provided us with access to specialists' points of view with respect to the findings of our next study (in Chapter 4) and some potential applications of our method and tool (in Section 7.2).



*Truth can only be found in one place: the code – Robert “Uncle Bob” C. Martin*

## STUDY II – ANALYZING THE EVOLUTION OF THE DEDICATION TO CONCERN

In the previous chapter, we evaluated our method and its strategies to measure our metric, Dedication to Concern (DtC). Now, we want to take advantage of DtC to look into how concerns evolve. We want to know which category of DtC (slight, moderate and high) is more common through the history of software systems.

We conjecture that the following are potential advantages of analyzing the evolution of DtC:

1. As our method enables maximizing the mining of concerns, researchers and practitioners may end up with massive datasets. So, it is valuable to consider ways to refine them, *e.g.*, filtering out samples of data whose association between concerns and source code lacks a significant DtC;
2. Enabling the definition of strategies to refactor source code. Developers may consider it relevant to increase the dedication of artifacts to concerns if AKS detects the relationship as being slight. For instance, they can modify the way how artifacts import and use third-party components to reach a higher DtC.

### 5.1 STUDY DEFINITION

Now, we describe how we conducted this study. We discuss the mining-analysis strategy which we applied to observe the evolution of DtC. We focus on the mining of concerns from a particular domain: non-relational database projects. Table 5.1 contains information about the systems that we mined to generate the study’s dataset. We used AKS to analyze 43369 files collected from the historical data of the systems.

Figure 5.1 summarizes the study’s activities. To fulfill the first activity we mined concerns and DtC following our method (described in Chapter 3), and produced a dataset that contains information about the non-relational databases. We split the dataset into

**Table 5.1** Analyzed Projects – Adapted from (CARVALHO; NOVAIS; MENDONÇA, 2020)

Domain	Project	Description	Period	Files
Graph	JanusGraph <sup>a</sup>	Highly scalable graph database	2017-04 - 2018-10	5668
	Neo4J <sup>b</sup>	High performance graph store with all the features expected from a robust database	2018-09 - 2018-12	26543
	Titan <sup>c</sup>	Database optimized for storing and querying large graphs	2012-06 - 2015-09	3576
Time Series	OpenTSDB <sup>d</sup>	Distributed, scalable TS database	2015-11 - 2018-12	1440
	KairosDb <sup>e</sup>	Fast distributed scalable TS database written on top of Cassandra	2015-11 - 2018-11	1884
	Heroic <sup>f</sup>	A scalable TS database based on Bigtable, Cassandra and Elasticsearch	2014-04 - 2019-02	4258

<sup>a</sup> <https://github.com/JanusGraph/janusgraph>

<sup>b</sup> <https://github.com/neo4j/neo4j>

<sup>c</sup> <https://github.com/thinkaurelius/titan>

<sup>d</sup> <https://github.com/OpenTSDB/opentsdb>

<sup>e</sup> <https://github.com/kairosdb/kairosdb>

<sup>f</sup> <https://github.com/spotify/heroic>

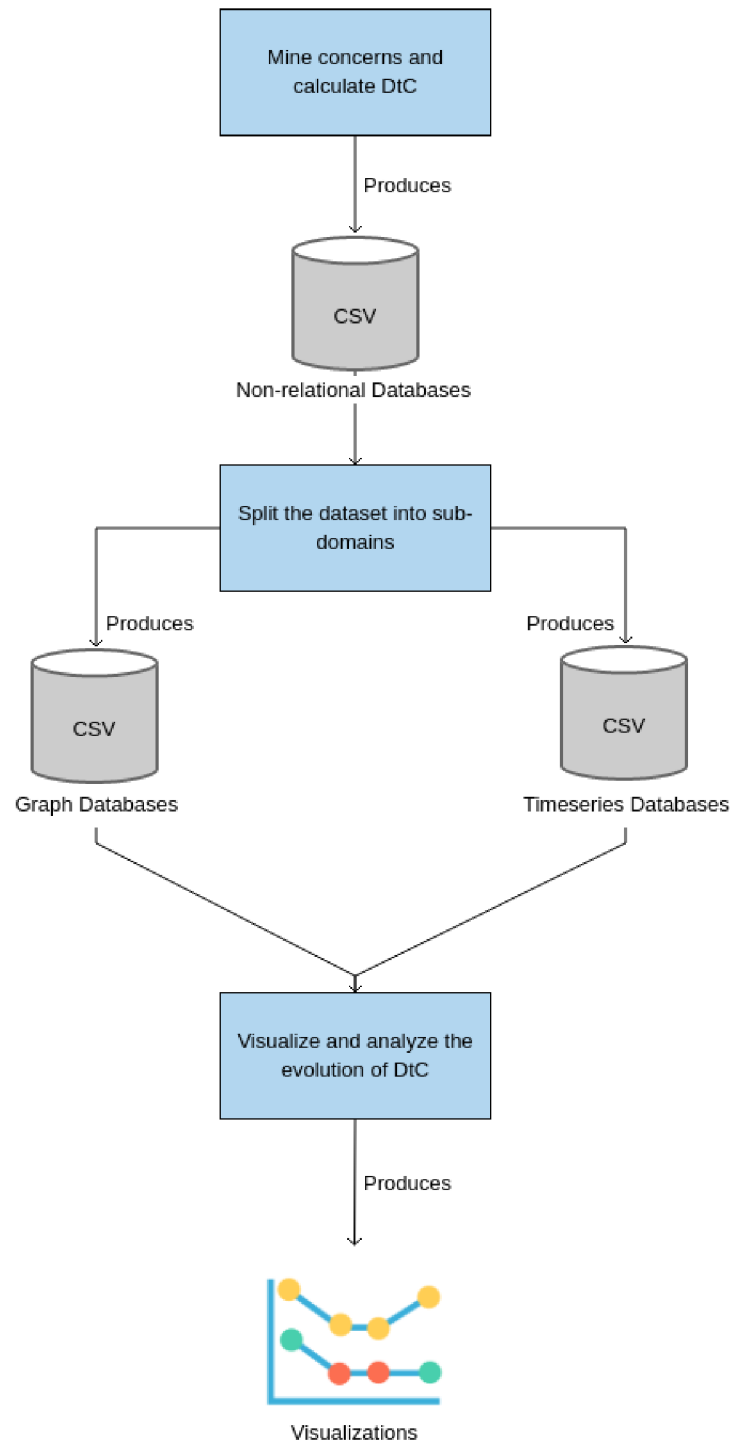
two subsets of data to observe the evolution of graph and time series databases separately. With each subset, we created charts to visualize and analyze the evolution of concerns and DtC. Then, we gathered insights from visualizations.

We dissociated the analysis of the graph databases from the analysis of the time series databases. The following is our rationale behind this decision:

1. Despite the fact that we can group graph and time series databases under the same domain (non-relational databases), their developers might still have different needs regarding the implementation of concerns;
2. Variations in the way on how developers implement concerns can be seen as a strategy to enhance our analyses: isolating each of the two types of database as top-level independent domains can empower us with extracting more concerns being shared by systems grouped as either graph or time series databases.

We used the mean Lines of Code (m-LOC) metric (KOCH, 2004) to compare the evolution of DtC as agglomerations<sup>1</sup>. Considering the fictitious example illustrated in Figure 5.2, AKS obtains the normalized value for “Test” and “Serialization” agglomerations (the circles of the figure) by calculating the mean-LOC (m-LOC) of artifacts

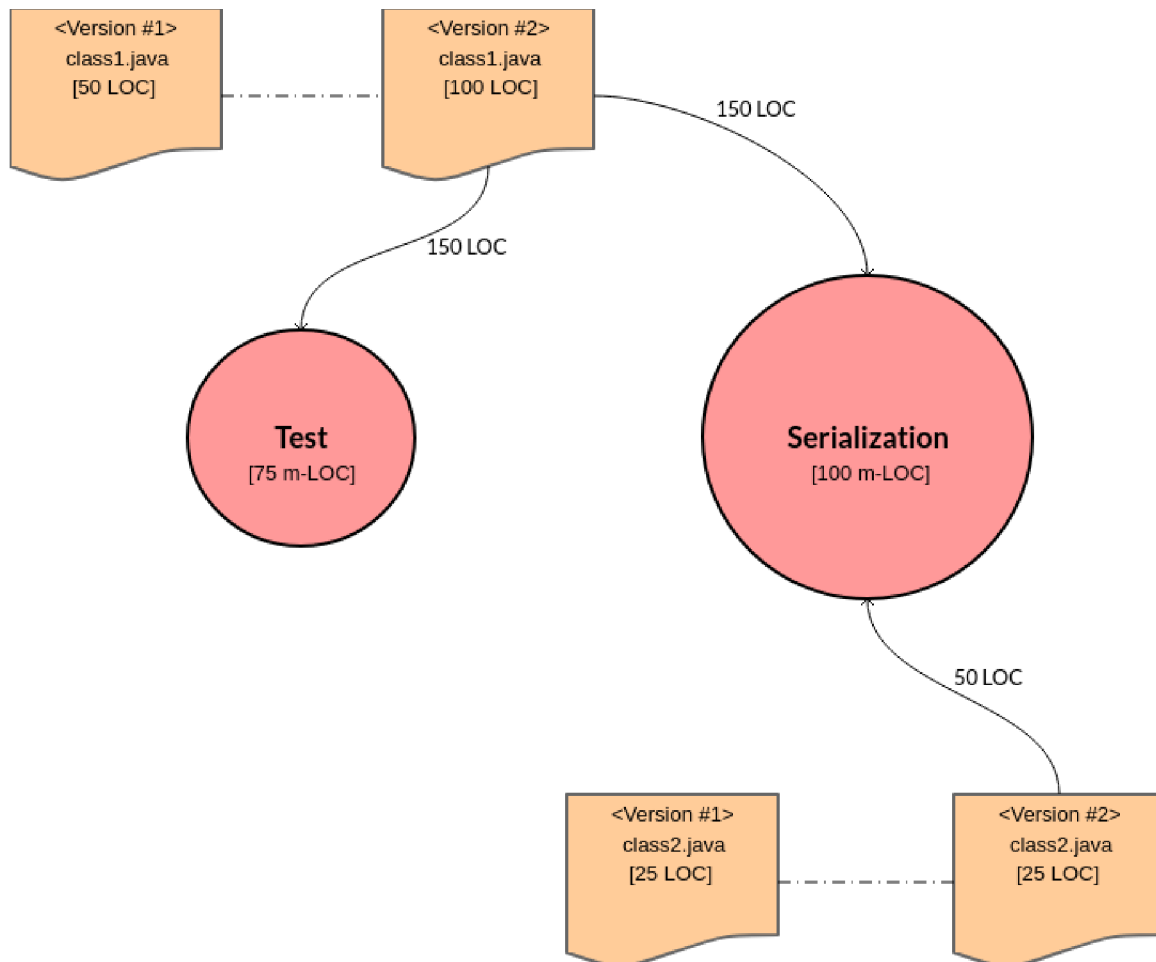
<sup>1</sup>More information about agglomerations can be found in Appendix C – Section C.1.2



**Figure 5.1** Summary of this Study

associated with the concerns through the software projects' evolution. For instance, AKS normalized the density of the "Test" agglomeration by dividing the sum of LOC measured from all instances of "class1.java" by the number of times the artifact appeared

during the evolution. As “class1.java” appeared twice, AKS sums the LOCs of the two versions of the artifact (50 and 100) and divides the result by two. Doing this adds 75 of m-LOC to “Test”. Similarly, AKS determines the density of “Serialization” (100) by summing the m-LOC of “class1.java” (75) and the m-LOC of “class2.java” (25). In the example, “class1.java” imports test-related components and “class2.java” executes test and serialization routines.



**Figure 5.2** DtC Agglomerations

We covered a number of six versions of each system. We selected two versions from the initial phases of the projects’ history, and two others from recent phases. We chose the last two ones from somewhere in between the initial and recent phases. This selection strategy allowed us to observe the evolution of concerns considering different moments of the software projects’ lifecycles.

## 5.2 RESULTS

Now, we present the results of our analysis about the evolution of DtC. Figures 5.3 and 5.4 follow the same pattern. They are made of a composition of charts. Each chart shows



the evolution of one particular concern through a series of six versions of the software projects listed in Table 5.1. The figures exhibit the number of version along the x-axis of the charts. The y-axis displays the m-LOC of agglomerations associated with the three categories of DtC: slight, moderate, and high.

Considering the agglomerations exhibited in Figure 5.3, it is possible to notice that they tend to remain slight through time. “Logging” is an example of this tendency. This is logical because developers often log several scattered routines of their systems, *i.e.*, “Logging” is not self-contained and is less likely to be tidily separated from other concerns throughout the codebase. It serves other concerns when it is necessary to register their execution steps and states. The source code snippet exhibited in Listing 5.1 exemplifies the auxiliary role of “Logging”.

AKS detected the “Logging” concern in Listing 5.1 because the artifact imports log-related components (lines 17 and 19, in yellow) and use them in the **mutateMany** method (line 61, in gray). Although logging is an important feature, its presence and use through many source code lines is not mandatory, *i.e.*, even if a developer embeds a logging component into a source code artifact, he/she may not need to log all routines performed by the artifact’s methods.

Like “Logging”, other concerns tend to remain slightly associated with artifacts because they often perform secondary roles during the development of the systems. For instance, “Service-Orientation” is added to software projects to externalize some functionalities which are served to client applications.

The source code excerpt exhibited in Listing 5.2 exemplifies how “Service-Orientation” is used in combination with other concerns to respond to client application’s requests. An import declaration (in line 7, in yellow) ties the source code with the “Service-Orientation” concern (in line 6, in green). Method **shouldRedirectToWebadminOnHtmlRequest** (in line 64, in gray) uses the imported component. The role of “Service-Orientation” is auxiliary to the central concern of the artifact: “Test” (specifically, testing of a Neo4j’s service discovery feature). The artifact is nearly entirely dedicated to perform testing routines because of the many test-related annotations preceding its methods (lines 21, 26, 33, 40, 47, and 64, in gray). As a result, AKS categorized the relationship between the artifact and “Service-Orientation” as slight.

**Listing 5.1** An artifact slightly dedicated to implement “Logging”

```

1  package org.janusgraph.diskstorage.locking.consistentkey;
2
3  import org.janusgraph.diskstorage.BackendException;
4  import org.janusgraph.diskstorage.StaticBuffer;
5  import org.janusgraph.diskstorage.BaseTransactionConfig;
6  import org.janusgraph.diskstorage.configuration.Configuration;
7  import org.janusgraph.diskstorage.configuration.MergedConfiguration;
8  import org.janusgraph.diskstorage.keycolumnvalue.*;
9  import org.janusgraph.diskstorage.locking.LockerProvider;
10 import org.janusgraph.diskstorage.util.StandardBaseTransactionConfig
    ;
11
12 import java.time.Duration;
13 import java.util.HashMap;

```

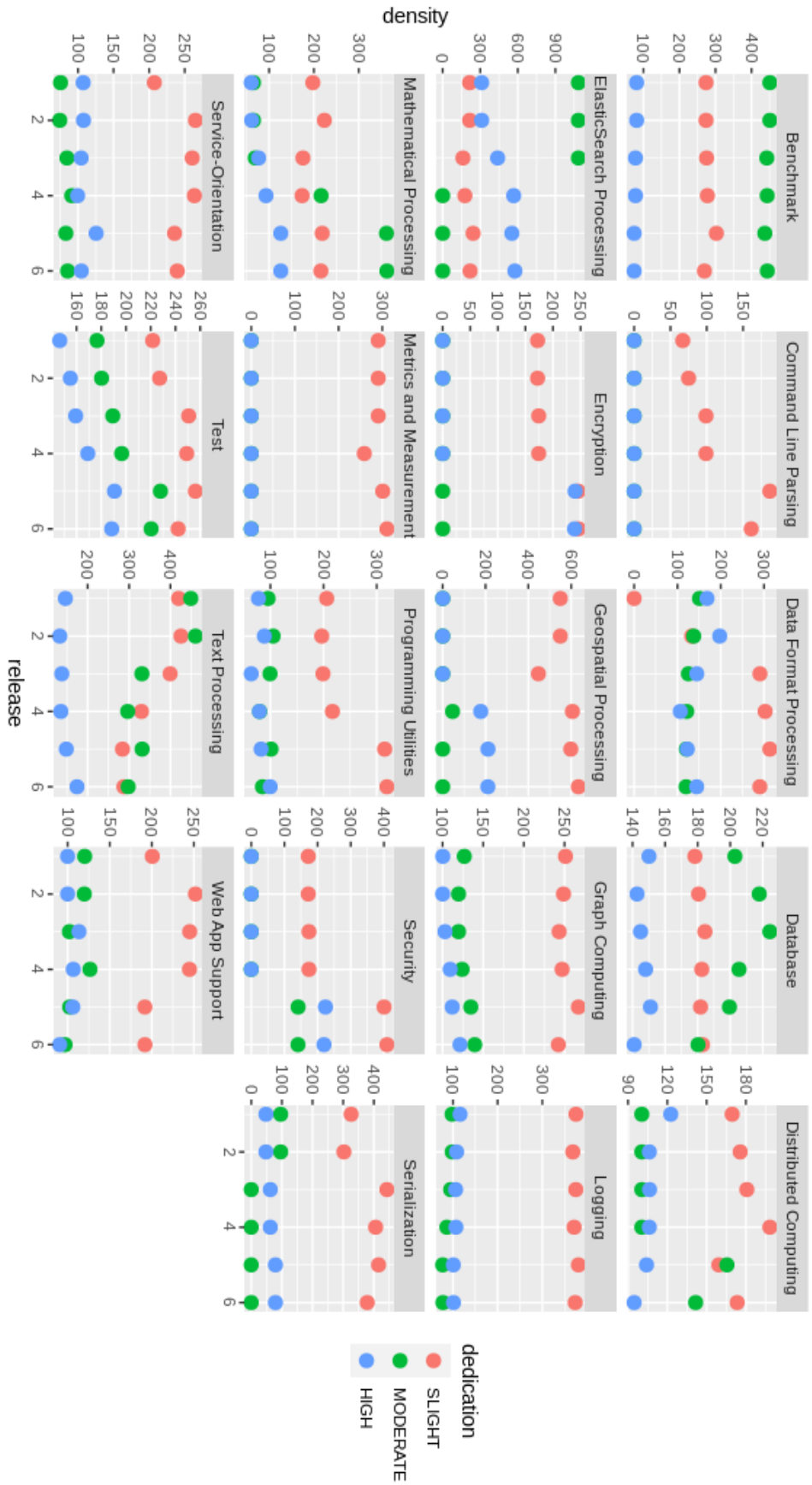


Figure 5.3 Graph Databases' Evolution of D+C

```

14 import java.util.Map;
15
16 @Concern(name="Logging")
17 import org.slf4j.Logger;
18 @Concern(name="Logging")
19 import org.slf4j.LoggerFactory;
20
21 /** @author Matthias Broecheler (me@matthiasb.com) */
22 public class ExpectedValueCheckingStoreManager extends
    KCVSManagerProxy {
23
24     private final String lockStoreSuffix;
25     private final LockerProvider lockerProvider;
26     private final Duration maxReadTime;
27     private final StoreFeatures storeFeatures;
28
29     private final Map<String , ExpectedValueCheckingStore> stores;
30
31     private static final Logger log =
32         LoggerFactory.getLogger(ExpectedValueCheckingStoreManager.
            class);
33
34     public ExpectedValueCheckingStoreManager(
35         KeyColumnValueStoreManager storeManager ,
36         String lockStoreSuffix ,
37         LockerProvider lockerProvider ,
38         Duration maxReadTime) {
39         super(storeManager);
40         this.lockStoreSuffix = lockStoreSuffix;
41         this.lockerProvider = lockerProvider;
42         this.maxReadTime = maxReadTime;
43         this.storeFeatures =
44             new StandardStoreFeatures.Builder(storeManager.getFeatures()
                ).locking(true).build();
45         this.stores = new HashMap<String , ExpectedValueCheckingStore>(6)
            ;
46     }
47
48     @Override
49     public synchronized KeyColumnValueStore openDatabase(String name)
        throws BackendException {
50         if (stores.containsKey(name)) return stores.get(name);
51         KeyColumnValueStore store = manager.openDatabase(name);
52         final String lockerName = store.getName() + lockStoreSuffix;
53         ExpectedValueCheckingStore wrappedStore =
54             new ExpectedValueCheckingStore(store , lockerProvider.
                getLocker(lockerName));
55         stores.put(name, wrappedStore);
56         return wrappedStore;
57     }
58

```

```

59 @Concern(name = "Logging")
60 @Override
61 public void mutateMany(
62     Map<String, Map<StaticBuffer, KCVMutation>> mutations, StoreTransaction txh)
63     throws BackendException {
64     ExpectedValueCheckingTransaction etx = (
65         ExpectedValueCheckingTransaction) txh;
66     boolean hasAtLeastOneLock = etx.prepareForMutations();
67     if (hasAtLeastOneLock) {
68         // Force all mutations on this transaction to use strong
69         // consistency
70         log.debug(
71             "Transaction {} holds one or more locks: writing using
72             consistent transaction {} due to held locks",
73             etx,
74             etx.getConsistentTx());
75         manager.mutateMany(mutations, etx.getConsistentTx());
76     } else {
77         log.debug(
78             "Transaction {} holds no locks: writing mutations using
79             store transaction {}",
80             etx,
81             etx.getInconsistentTx());
82         manager.mutateMany(mutations, etx.getInconsistentTx());
83     }
84 }
85
86 @Override
87 public ExpectedValueCheckingTransaction beginTransaction(
88     BaseTransactionConfig configuration)
89     throws BackendException {
90     // Get a transaction without any guarantees about strong
91     // consistency
92     StoreTransaction inconsistentTx = manager.beginTransaction(
93         configuration);
94
95     // Get a transaction that provides global strong consistency
96     Configuration customOptions =
97         new MergedConfiguration(
98             storeFeatures.getKeyConsistentTxConfig(), configuration.
99             getCustomOptions());
100     BaseTransactionConfig consistentTxCfg =
101         new StandardBaseTransactionConfig.Builder(configuration)
102             .customOptions(customOptions)
103             .build();
104     StoreTransaction strongConsistentTx = manager.beginTransaction(
105         consistentTxCfg);
106
107     // Return a wrapper around both the inconsistent and consistent
108     // store transactions
109     ExpectedValueCheckingTransaction wrappedTx =

```

```

100         new ExpectedValueCheckingTransaction(inconsistentTx ,
101             strongConsistentTx , maxReadTime);
102     }
103
104     @Override
105     public StoreFeatures getFeatures() {
106         return storeFeatures;
107     }
108 }

```

**Listing 5.2** An artifact slightly dedicated to implement “Service-Orientation”

```

1  package org.neo4j.server.rest;
2
3  import java.util.Map;
4  import javax.ws.rs.core.MediaType;
5
6  @Concern(name="Service-Orientation")
7  import com.sun.jersey.api.client.Client;
8  import org.junit.Before;
9  import org.junit.Test;
10
11 import org.neo4j.server.rest.domain.JsonHelper;
12
13 import static org.hamcrest.MatcherAssert.assertThat;
14 import static org.hamcrest.Matchers.containsString;
15 import static org.junit.Assert.assertEquals;
16 import static org.junit.Assert.assertNotNull;
17 import static org.junit.Assert.assertTrue;
18
19 public class DiscoveryServiceDocIT extends
20     AbstractRestFunctionalTestBase {
21     @Before
22     public void cleanTheDatabase() {
23         cleanDatabase();
24     }
25
26     @Test
27     public void shouldRespondWith200WhenRetrievingDiscoveryDocument() throws Exception {
28         JaxRsResponse response = getDiscoveryDocument();
29         assertEquals(200, response.getStatus());
30         response.close();
31     }
32
33     @Test
34     public void shouldGetContentLengthHeaderWhenRetrievingDiscoveryDocument() throws Exception {
35         JaxRsResponse response = getDiscoveryDocument();
36         assertNotNull(response.getHeaders().get("Content-Length"));
37         response.close();
38     }
39 }

```

```

38
39 @Test
40 public void shouldHaveJsonMediaTypeWhenRetrievingDiscoveryDocument() throws Exception {
41     JaxRsResponse response = getDiscoveryDocument();
42     assertThat(response.getType().toString(), containsString(
43         MediaType.APPLICATION_JSON));
44     response.close();
45 }
46 @Test
47 public void shouldHaveJsonDataInResponse() throws Exception {
48     JaxRsResponse response = getDiscoveryDocument();
49
50     Map<String, Object> map = JsonHelper.jsonToMap(response.
51         getEntity());
52
53     String managementKey = "management";
54     assertTrue(map.containsKey(managementKey));
55     assertNotNull(map.get(managementKey));
56
57     String dataKey = "data";
58     assertTrue(map.containsKey(dataKey));
59     assertNotNull(map.get(dataKey));
60     response.close();
61 }
62 @Concern(name = "Service-Orientation")
63 @Test
64 public void shouldRedirectToWebadminOnHtmlRequest() throws Exception {
65     Client nonRedirectingClient = Client.create();
66     nonRedirectingClient.setFollowRedirects(false);
67
68     JaxRsResponse clientResponse =
69         new RestRequest(null, nonRedirectingClient)
70             .get(server().baseUri().toString(), MediaType.
71                 TEXT_HTML_TYPE);
72
73     assertEquals(303, clientResponse.getStatus());
74 }
75 private JaxRsResponse getDiscoveryDocument() throws Exception {
76     return new RestRequest(server().baseUri()).get();
77 }
78 }

```

We could argue that “Test” is not self-contained because developers add tests components to evaluate and validate distinct features of the systems. Thus, “Test” should always appear in combination with other concerns. However, in Figure 5.3 we show that AKS detected situations in which the DtC of “Test” evolves as moderate and high. This comes from developers frequently specializing test routines to deal with specific sets of

functionalities. The specialization stems from a good practice related to tests automation: single responsibility. This principle is better achieved when test routines focus on either very few or on one single behavior of a system (RUNESON, 2006)(BOWES et al., 2017). Listing 5.3 exemplifies this principle.

The import declarations (lines 4, 6, 8, 10, and 16, in yellow) link the artifact with the “Test” concern. The two methods of the source code, **shouldBeAbleToAddCauses** (line 21, in gray) and **stackTraceShouldContainAllCauses** (line 34, in gray), focus on testing one single behavior of Neo4j: processing the content of exception messages. Importing and using many test-related components to evaluate this specific behavior caused AKS to detect that the artifact is highly dedicated to implement “Test”.

**Listing 5.3** An artifact slightly dedicated to implement “Test”

```

1 package org.neo4j.kernel.impl.util;
2
3 @Concern(name="Test")
4 import static org.hamcrest.CoreMatchers.is;
5 @Concern(name="Test")
6 import static org.hamcrest.CoreMatchers.not;
7 @Concern(name="Test")
8 import static org.hamcrest.CoreMatchers.nullValue;
9 @Concern(name="Test")
10 import static org.junit.Assert.assertThat;
11
12 import java.io.ByteArrayOutputStream;
13 import java.io.PrintWriter;
14
15 @Concern(name="Test")
16 import org.junit.Test;
17
18 public class TestMultipleCauseException {
19     @Concern(name="Test")
20     @Test
21     public void shouldBeAbleToAddCauses() {
22         Throwable cause = new Throwable();
23         MultipleCauseException exception = new MultipleCauseException("
24             Hello", cause);
25         assertThat(exception.getMessage(), is("Hello"));
26         assertThat(exception.getCause(), is(cause));
27         assertThat(exception.getCauses(), is(not(nullValue())));
28         assertThat(exception.getCauses().size(), is(1));
29         assertThat(exception.getCauses().get(0), is(cause));
30     }
31
32     @Concern(name="Test")
33     @Test

```

```

34  public void stackTraceShouldContainAllCauses() {
35      Throwable cause1 = new Throwable("Message 1");
36      MultipleCauseException exception = new MultipleCauseException("
          Hello", cause1);
37
38      Throwable cause2 = new Throwable("Message 2");
39      exception.addCause(cause2);
40
41      ByteArrayOutputStream baos = new ByteArrayOutputStream();
42      PrintWriter out = new PrintWriter(baos);
43
44      // When
45      exception.printStackTrace(out);
46      out.flush();
47      String stackTrace = baos.toString();
48
49      // Then
50      assertThat(
51          "Stack trace contains exception one as cause.",
52          stackTrace.contains("Caused by: java.lang.Throwable: Message
              1"),
53          is(true));
54      assertThat(
55          "Stack trace contains exception one as cause.",
56          stackTrace.contains("Also caused by: java.lang.Throwable:
              Message 2"),
57          is(true));
58  }
59  }

```

Figure 5.4 exhibits the evolution of time series databases’ DtC. It shows some resemblances with the graph databases: (i) some concerns tend to remain slightly associated with artifacts (*e.g.*, “Logging”, “Dependency Injection”, “Process Execution”) as they tend to play auxiliary roles; and (ii) some concerns, like “Test”, are exceptions because developers of time series databases specialize some artifacts to run tests.

In a way similar to “Test”, “Database” is another concern that developers separate in specialized artifacts. The source code snippet exhibited in Listing 5.4 illustrates this situation. The import declarations (lines 4, 6, and 8, highlighted in yellow) associate the artifact with the “Database” concern. The majority of the methods (in gray) is dedicate to execute database-related tasks: **runQuery** (line 44), **runQuery**’s overload (line 79), **close** (line 134), **getArrayList** (line 148), **getArrayList**’s overload (line 179), **getRecord** (line 198), **getOnlyRecord** (line 214), and **next** (line 234).

**Listing 5.4** An artifact moderately dedicated to implement “Database”

```

1  package org.kairosdb.datastore.h2.orm;
2
3  @Concern(name="Database")
4  import org.agileclick.genorm.runtime.GenOrmException;
5  @Concern(name="Database")

```



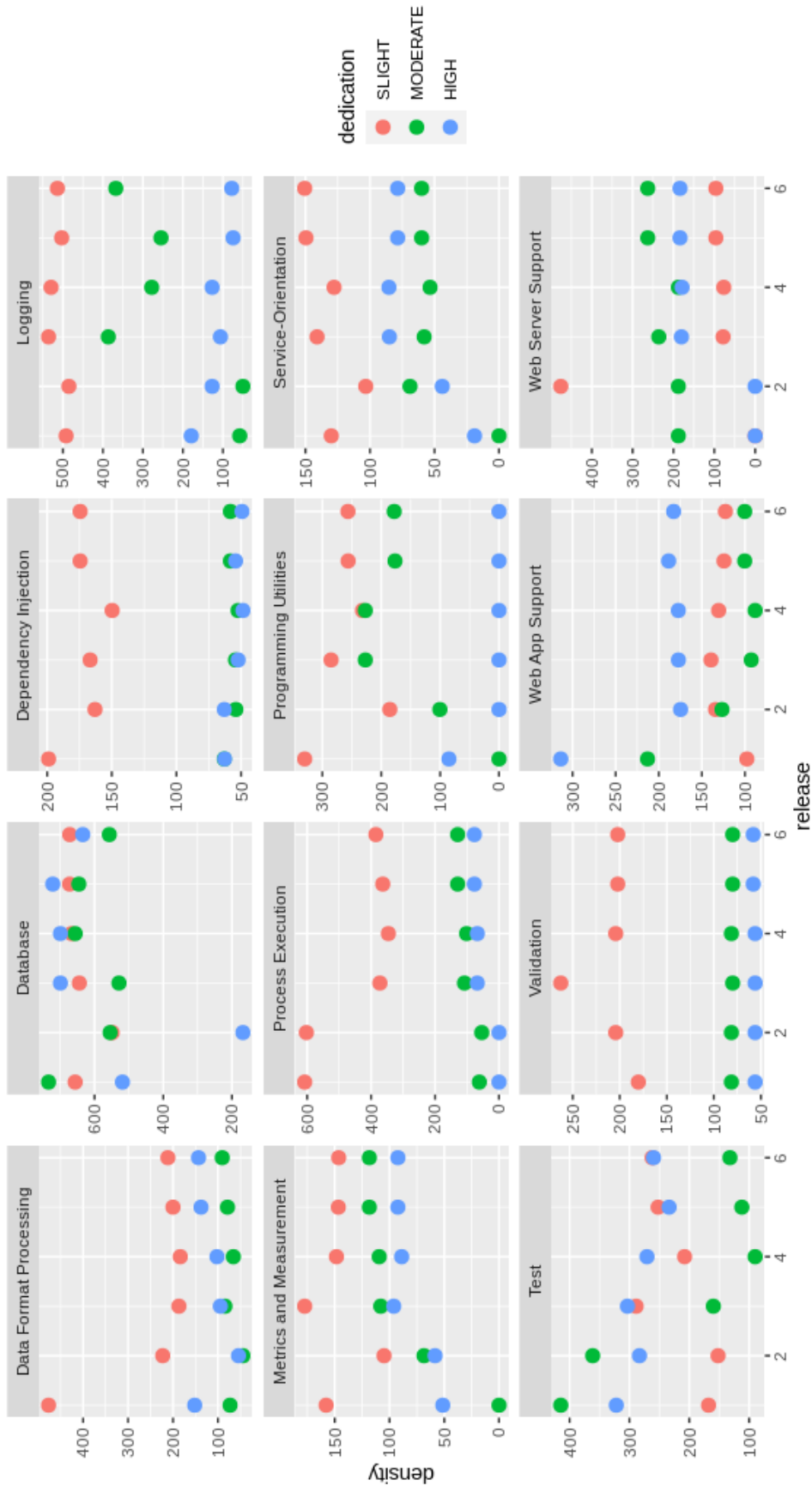


Figure 5.4 Time Series Databases' Evolution of DtC

```

6  import org.agileclick.genorm.runtime.GenOrmQueryRecord;
7  @Concern(name="Database")
8  import org.agileclick.genorm.runtime.GenOrmQueryResultSet;
9  import org.slf4j.Logger;
10 import org.slf4j.LoggerFactory;
11 import org.xml.sax.Attributes;
12 import org.xml.sax.ContentHandler;
13
14 import java.util.ArrayList;
15 import java.util.HashMap;
16 import java.util.List;
17 import java.util.Map;
18
19 /** */
20 public class MetricNamesPrefixQuery extends org.agileclick.genorm.
    runtime.SQLQuery {
21     private static final Logger s_logger =
22         LoggerFactory.getLogger(MetricNamesPrefixQuery.class.getName()
23             );
24
25     public static final String QUERY_NAME = "metric_names_prefix";
26     public static final String QUERY = "select distinct \"name\" from
27         metric where \"name\" like ?";
28     private static final int ATTRIBUTE_COUNT = 1;
29     private static Map<String, Integer> s_attributeIndex;
30     private static String[] s_attributeNames = {"name"};
31
32     static {
33         s_attributeIndex = new HashMap<String, Integer>();
34         for (int I = 0; I < ATTRIBUTE_COUNT; I++) s_attributeIndex.put(
35             s_attributeNames[I], I);
36     }
37
38     private boolean m_serializable;
39     ...
40
41     public void serializeQuery(ContentHandler ch, String tagName)
42         throws org.xml.sax.SAXException {
43         ...
44     }
45
46     @Concern(name="Database")
47     public ResultSet runQuery(String prefix) {
48         java.sql.PreparedStatement genorm_statement = null;
49         try {
50             String genorm_query = QUERY;
51
52             genorm_statement =
53                 org.kairosdb.datastore.h2.orm.GenOrmDataSource.
54                     prepareStatement(genorm_query);

```

```

51     genorm_statement.setString(1, prefix);
52
53     long genorm_queryTimeStart = 0L;
54     if (s_logger.isInfoEnabled()) {
55         genorm_queryTimeStart = System.currentTimeMillis();
56     }
57
58     java.sql.ResultSet genorm_resultSet = genorm_statement.
        executeQuery();
59
60     if (genorm_queryTimeStart != 0L) {
61         long genorm_quryTime = System.currentTimeMillis() -
            genorm_queryTimeStart;
62         s_logger.info(String.valueOf(genorm_quryTime));
63     }
64
65     ResultSet genorm_ret = new SQLResultSet(genorm_resultSet,
        genorm_statement, genorm_query);
66
67     return (genorm_ret);
68 } catch (java.sql.SQLException sqle) {
69     try {
70         if (genorm_statement != null) genorm_statement.close();
71     } catch (java.sql.SQLException sqle2) {
72     }
73
74     throw new GenOrmException(sqle);
75 }
76 }
77
78 @Concern(name="Database")
79 public ResultSet runQuery() {
80     java.sql.PreparedStatement genorm_statement = null;
81     try {
82         String genorm_query = QUERY;
83
84         genorm_statement =
85             org.kairosdb.datastore.h2.orm.GenOrmDataSource.
                prepareStatement(genorm_query);
86         genorm_statement.setString(1, m_prefix);
87
88         long genorm_queryTimeStart = 0L;
89         if (s_logger.isInfoEnabled()) {
90             genorm_queryTimeStart = System.currentTimeMillis();
91         }
92
93         java.sql.ResultSet genorm_resultSet = genorm_statement.
            executeQuery();
94
95         if (genorm_queryTimeStart != 0L) {
96             long genorm_quryTime = System.currentTimeMillis() -
                genorm_queryTimeStart;

```

```

97         s_logger.info(String.valueOf(genorm_queryTime));
98     }
99
100    ResultSet genorm_ret = new SQLResultSet(genorm_resultSet,
101                                           genorm_statement, genorm_query);
102
103    return (genorm_ret);
104 } catch (java.sql.SQLException sqle) {
105     try {
106         if (genorm_statement != null) genorm_statement.close();
107     } catch (java.sql.SQLException sqle2) {
108     }
109
110     throw new GenOrmException(sqle);
111 }
112
113 public interface ResultSet extends GenOrmQueryResultSet<
114     MetricNamesPrefixData> {
115     ...
116 }
117
118 private class SQLResultSet implements ResultSet {
119     private java.sql.ResultSet m_resultSet;
120     private java.sql.Statement m_statement;
121     private String m_query;
122     private boolean m_onFirstResult;
123
124     // need to pass in the statement so it can be closed after the
125     // result set
126     protected SQLResultSet(
127         java.sql.ResultSet resultSet, java.sql.Statement statement,
128         String query) {
129         ...
130     }
131
132     /**
133     * Closes any underlying java.sql.Result set and java.sql.
134     * Statement that was used to create this
135     * results set.
136     */
137     @Concern(name="Database")
138     public void close() {
139         try {
140             m_resultSet.close();
141             m_statement.close();
142         } catch (java.sql.SQLException sqle) {
143             throw new GenOrmException(sqle);
144         }
145     }
146 }
147
148 /**

```

```

144     * Returns the results as an ArrayList of Record objects. The
145     * Result set is closed within this
146     * call
147     */
148     @Concern(name="Database")
149     public List<MetricNamesPrefixData> getArrayList(int maxRows) {
150         ArrayList<MetricNamesPrefixData> results = new ArrayList<
151             MetricNamesPrefixData >();
152         int count = 0;
153         try {
154             if (m_onFirstResult) {
155                 count++;
156                 results.add(new MetricNamesPrefixData(
157                     MetricNamesPrefixQuery.this, m_resultSet));
158             }
159             while (m_resultSet.next() && (count < maxRows)) {
160                 count++;
161                 results.add(new MetricNamesPrefixData(
162                     MetricNamesPrefixQuery.this, m_resultSet));
163             }
164             if (m_resultSet.next())
165                 throw new GenOrmException(
166                     "Bound of " + maxRows + " is too small for query [" +
167                     m_query + "]"");
168             } catch (java.sql.SQLException sqle) {
169                 throw new GenOrmException(sqle);
170             }
171         }
172         close();
173         return (results);
174     }
175     /**
176     * Returns the results as an ArrayList of Record objects. The
177     * Result set is closed within this
178     * call
179     */
180     @Concern(name="Database")
181     public List<MetricNamesPrefixData> getArrayList() {
182         ArrayList<MetricNamesPrefixData> results = new ArrayList<
183             MetricNamesPrefixData >();
184         try {
185             if (m_onFirstResult)
186                 results.add(new MetricNamesPrefixData(
187                     MetricNamesPrefixQuery.this, m_resultSet));
188             while (m_resultSet.next())

```

```

187         results.add(new MetricNamesPrefixData(
188             MetricNamesPrefixQuery.this, m_resultSet));
189     } catch (java.sql.SQLException sqle) {
190         throw new GenOrmException(sqle);
191     }
192     close();
193     return (results);
194 }
195
196 /** Returns the current record in the result set */
197 @Concern(name="Database")
198 public MetricNamesPrefixData getRecord() {
199     MetricNamesPrefixData ret = null;
200     try {
201         ret = new MetricNamesPrefixData(MetricNamesPrefixQuery.this,
202             m_resultSet);
203     } catch (java.sql.SQLException sqle) {
204         throw new GenOrmException(sqle);
205     }
206     return (ret);
207 }
208
209 /**
210  * This call expects only one record in the result set. If
211  * multiple records are found an
212  * exception is thrown. The ResultSet object is automatically
213  * closed by this call.
214  */
215 @Concern(name="Database")
216 public MetricNamesPrefixData getOnlyRecord() {
217     MetricNamesPrefixData ret = null;
218
219     try {
220         if (m_resultSet.next())
221             ret = new MetricNamesPrefixData(MetricNamesPrefixQuery.
222                 this, m_resultSet);
223
224         if (m_resultSet.next())
225             throw new GenOrmException(
226                 "Multiple rows returned in call from
227                 MetricNamesPrefixQuery.ResultSet.getOnlyRecord");
228     } catch (java.sql.SQLException sqle) {
229         throw new GenOrmException(sqle);
230     }
231     close();
232     return (ret);
233 }

```

```

232     /** Returns true if there is another record in the result set.
233         */
234     @Concern(name="Database")
235     public boolean next() {
236         boolean ret = false;
237         m_onFirstResult = true;
238         try {
239             ret = m_resultSet.next();
240         } catch (java.sql.SQLException sqle) {
241             throw new GenOrmException(sqle);
242         }
243         return (ret);
244     }
245 }
246
247 public class Record implements GenOrmQueryRecord, Attributes {
248     ...
249 }
250 }

```

From our observations, we conjecture that there are concerns which are natural accessories to others, like “Service-Orientation” and “Logging”, but there are other concerns that developers may decide to isolate within specialized artifacts. “Test” and “Database” fits in this category.

Some of the charts in Figures 5.3 show non-uniform patterns. For example, the evolution of “ElasticSearch Processing” displays an abrupt decline in the number of moderate DtC from the third to fourth versions. Similar cases of non-uniformity can be seen in “Distributed Computing” and “Mathematical Processing” charts. This is mainly caused by our decision of choosing versions from the databases’ historical data that are not perfectly sequential through time. As a consequence, we may not have covered some intervals of time during which highly impacting refactorings happened. Sudden raises and drops in the charts can be caused by refactorings influencing how we measure DtC. The same can be said about some concerns exhibited in 5.4, *e.g.*, “Database”.

### 5.3 THREATS TO VALIDITY

We recognize the following as threats that can affect this study:

**Construct validity:** we decided to analyze graph and time series databases in isolation as a way to observe how DtC evolves after grouping systems according to their respective domains. However, we wonder if advancing in splitting the databases further would impact our analysis even more. Perhaps, among the time series and graph databases there is a set of projects that share more similar characteristics and concerns with each other. Thus, separating the historical data of systems through deeper branches of software domains would give us more variations in the way how each group of software implements concerns;

**Internal validity:** we see another limitation in not having selected sequential ver-

sions of the database projects. While we trust that the results which we presented in Section 5.2 are correct, we believe that we could end up with more smooth distributions of data if we mined/analyzed systems’ versions that are continuous through time. Additionally, mining older versions of the systems may have filled our dataset with unstable data, *i.e.*, mining information from software projects’ initial versions can reveal associations between source code artifacts and concerns from a time when developers were not sure about which concerns to implement and/or how to implement them;

**External validity:** we cannot condition a generic evaluation of DtC to the cases that we examined here. This means: we cannot guarantee that our findings regarding how DtC evolves can embrace the totality of software projects that exist. Our findings are constrained to the non-relational database projects that we inspected. Fortunately, we designed our method and ASK to be expandable, so they can be used to process other software projects and domains.

**Conclusion validity:** with this study we gathered some extra insights about the evolution of concerns via the analysis of our metric, DtC. Nonetheless, we believe that we still have much more to uncover regarding the relationship between source code artifacts and concerns through systems’ historical data. This requires defining a set of research questions and using the mining and analysis capabilities of AKS to answer them. Only then we will be able to assert the veracity of our insights regarding how DtC evolves with more precision.

## 5.4 DISCUSSION

With this study, we expand the initial ones which we presented in Chapter 4 and Appendix C. We chose software projects from one specific domain, non-relational databases, and examined them separately, as graph and time series databases. Then, we observed how our metric, DtC, evolved.

We noticed a tendency of DtC to remain slight through the historical data of the databases. Our results point to a general concept about concerns obtained from third-party components: they are likely to play secondary/auxiliary roles in the making of software projects. This is the case of the “Logging” concern that we exemplified in Section 5.2. Logging all of an artifact’s routines is not mandatory. Consequently, “Logging” is not prone to branch deep through many lines of source code.

The aforementioned tendency is not free from exceptions though. We found examples of source code artifacts moderately/highly dedicated to implement “Test” and “Database”. Such examples reveal that developers may specialize artifacts to deal with few features sometimes. Thus, although the separation of concerns is desirable, it is not always achievable. This is not entirely reprehensible if the particular nature of each concern is considered. While developers specialize artifacts to perform tests, they do not feel like doing the same when they need to log the activity of routines.

In Section 4.4.4, we described an unstructured interview that we conducted with the specialists that participated in our action research study. We took the opportunity to show the results of this study to them. They see as natural that concerns obtained from third-party components’ metadata tend to play auxiliary roles in the making of systems.



They also agreed with our observations that the relationship between some concerns and source code artifacts tend to be exceptional sometimes, and artifacts may focus on implementing a single or few concerns to address a need, *e.g.*, dedicating an artifact to test specific features.

## 5.5 CONCLUSION

Soon after we saw the possibility of identifying and processing concerns from the metadata of third-party components, we perceived that it was crucial to address some unfavorable circumstances. Specifically, our method is dependant on the import declarations that developers add to source code to inject components. As AKS uses the declarations to associate concerns with systems' artifacts, it is important to differentiate the distinct situations in which concerns relate to software projects at code level. In Section 3.2, we described how these situations made us define a new metric: Dedication to Concern (DtC).

After we evaluated and refined our method and reached a moderate agreement between raters, we took the opportunity to understand how DtC evolves through time (the main purpose of this chapter). As a result, we found out that some concerns are not finely separated/isolated in source code artifacts. This is logical regarding the characteristics of some concerns, *i.e.*, "Logging".

It is necessary to expand this study in determining which concerns obtained from third-party components are either more or less inclined to be separated in specialized artifacts. This can shed a light on the different strategies that developers follow when they either join or separate concerns.



*People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones – Donald Knuth*

## DISSEMINATION OF OUR RESEARCH

Research dissemination can be defined as “the communication or spread of new or existing knowledge through a planned or systematic process and implies that information is tailored for an intended target audience” (LOMAS, 1993)(LAWRENCE, 2006). “Dissemination” requires an intense flow of refined information from its source (the results of a research, for instance) to an identifiable public. Plus, if the information is adequately presented, it can raise awareness about concepts and consensus (LOMAS, 1993). As we desire that our method becomes a standard in the field of concerns identification, we have taken measures to format, organize and publicize the body of knowledge that we achieved from our studies.

Disseminating our results and contributions relies on three main actions: (i) making our tool’s source code available for reuse in a public repository (explained in Section 6.1), (ii) publicizing our studies’ datasets (more information in Section 6.2), and (iii) writing and publishing papers (listed in Section 6.3). We provide further details about the dissemination of our work through the next sub-sections.

### 6.1 REUSE OF OUR TOOL – AKS

Reuse is a way to capitalize on existing systems while increasing the quality of the final software product (BARROS-JUSTO et al., 2018). Regarding our tool, Architectural Knowledge Suite (AKS), we envision two types of reuse: (i) reuse by researchers and practitioners who want to analyze the presence of concerns in software projects and (ii) reuse by other developers who desire to either adapt AKS according to their needs or embed our mining and analysis routines into their own tools.

As a way to address reuse while fulfilling dissemination, we took another concept into account: Open Source Software (OSS). OSS can be defined as the source code that is shared to enable learning, modifying, and extending other software projects under some licensing guidelines (ELLIS; BELLE, 2009)(GANDHI; GONDWAL; TANDON, 2017).

By disclosing AKS source code, we aim at achieving a common advantage expected from OSS (FRANCO-BEDOYA et al., 2017): spreading of technologies, allowing users to use freely publicly available software and to incorporate third-party source code into their implementations. We have made the source code of ASK open and available for reuse at:

<https://gitlab.com/luispscarvalho/AKS>

Usually, OSS projects are initiated by an individual or a small group of people who want to fill a technological gap (FRANCO-BEDOYA et al., 2017), but, as the development evolves, the involvement of other people is desirable. Ideally, rather than a single corporate entity owning the software, a broad community of volunteers should determine which contributions are accepted into the codebase and where the OSS project must head to (RIEHLE, 2007). Investing in knowledge sharing is a way to attract the attention of potential contributors. This can be (partially) achieved by building up and organizing the memory and content related to OSS systems (HEMETSBERGER; REINHARDT, 2004). We have materialized such approach by adding some informative/descriptive artifacts to AKS' repository:

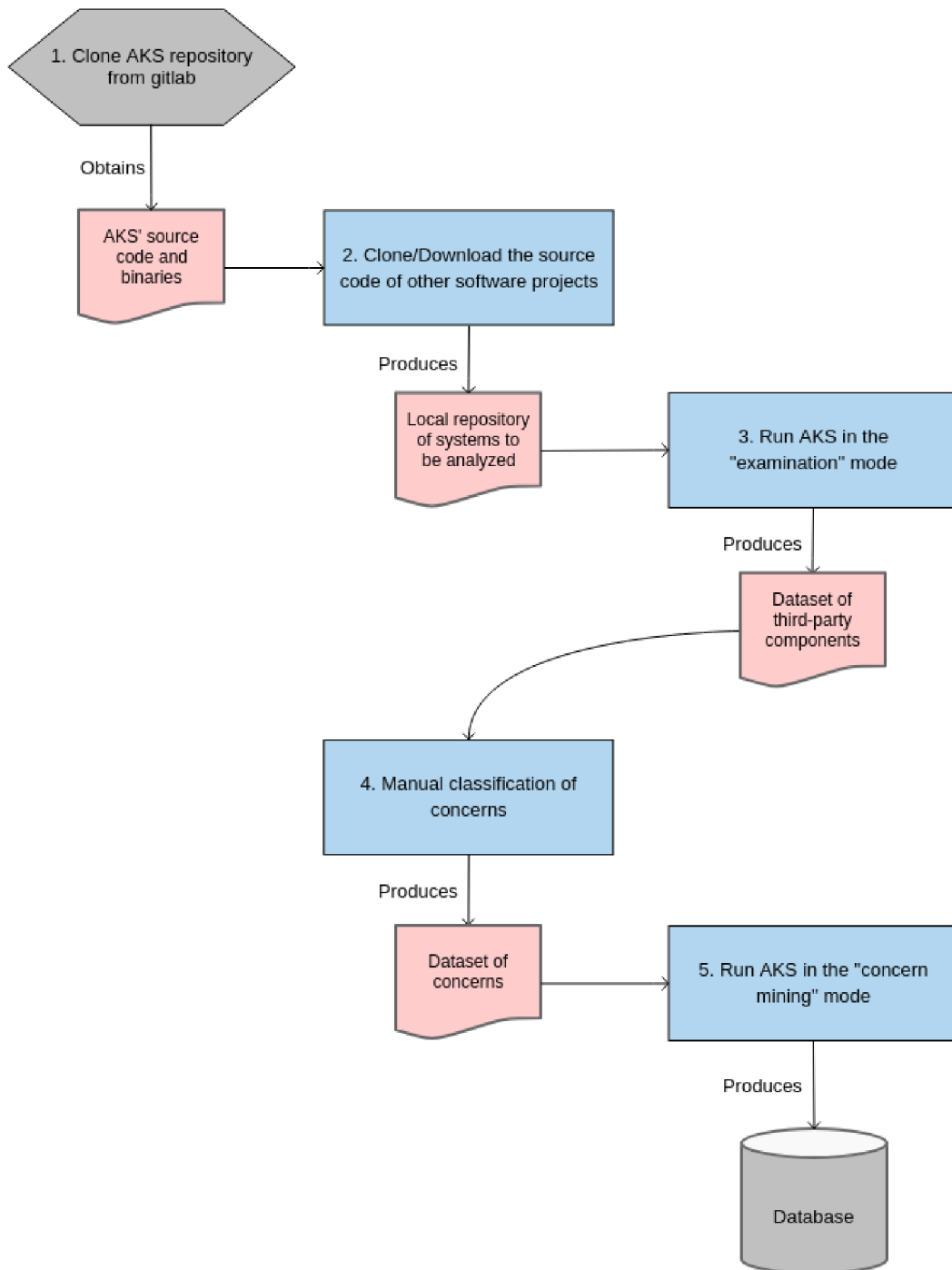
1. Tutorials: in the form of documents (PDF) and videos. We included a README file<sup>1</sup> in AKS' repository to indicate the location of our tutorials;
2. Advertising about our tool in our publications: we have added sections dedicated to describing our tool to each of our papers (listed in Section 6.3);
3. Presenting our tool: whenever we are given a chance, we present our tool. For instance, during events that we participate in.

We suggest the workflow shown in Figure 6.1 to reuse AKS. Firstly, AKS must be cloned/downloaded from the its repository (Activity 1). Second step involves choosing and downloading software projects which AKS will mine concerns from (Activity 2). AKS must be executed in the "examination" mode (Activity 3) to parse and analyze systems' third-party components' metadata before performing the actual mining of concerns. It will also fetch extra information about components (*e.g.*, categories, tags) from online repositories (*e.g.*, MVNRepository). As explained in Section 3.1.2, our method is dependant on manual interventions to associate components with fitting concerns. So, it is necessary to define them after checking the information that AKS acquires from components' metadata and from online repositories (Activity 4). After reaching consensus about the concerns, AKS can finally run in the "concern mining" mode to generate its database (Activity 5).

We adopted the *GNU General Public License v3.0* or GPL3<sup>2</sup> to ensure a permissive ruse of AKS. GPL has flourished as an OSS license for three reasons (TSAI, 2008): (i) it appeals to many software developers and users as a guideline to support collaboration over large geographical distances, (ii) software licensed under GPL is usually provided

<sup>1</sup><https://gitlab.com/luispscarvalho/AKS/-/blob/master/README.md>

<sup>2</sup><https://gitlab.com/luispscarvalho/AKS/-/blob/master/LICENSE>



**Figure 6.1** Reuse of our tool, Architectural Knowledge Suite

with no cost, and (iii) its reciprocal requirement that any distributed modifications must themselves be licensed under the GPL helps to perpetuate both the license and the licensed software.

## 6.2 REPLICATION PACKAGES

Making replication packages available for other researchers and practitioners is another way to accomplish research dissemination. Reuse of replication packages has emerged as a good research practice mainly due to (GÓMEZ; JURISTO; VEGAS, 2014)(CAMPBELL; STANLEY, 2015)(ALVAREZ; KEY; NÚÑEZ, 2018): (i) cloud-based computing making data and code sharing trivial, (ii) graduate programs training students to consider replications packages in their workflows, (iii) researchers perceiving that providing data, source code and other materials can boost their visibility and increase citations, (iv) journals and funding agencies requiring that research material be made available via publications, (v) a highly publicized number of issues regarding research transparency, and (vi) the need of replicating studies at other times and under different conditions before they can produce an established piece of knowledge.

As a way to organize and publicize our datasets, we created another repository in gitlab to accomodate them<sup>3</sup>. As we did with AKS, we decided to license our datasets under GLP3.

Next sub-sections describe our studies' replication packages. In general, each package contains: (i) spreadsheets which we filled with concerns-related data that ASK mined from software projects' repositories, (ii) analysis scripts that can be used to process the spreadsheets and generate visualizations, and (iii) extra documents/files that we considered important to contextualize the application of our method and the reuse of the packages (*i.e.*, tutorials, guides).

### 6.2.1 Study I's Replication Package

Our first replication package encapsulates a dataset about our action research study (Chapter 4). With this package, it is possible to reproduce/reevaluate the study's rounds of agreement (discussed through Section 4.4). Its spreadsheets also contain data about the measurement of our metric, Dedication to Concern (DtC). The package is located here:

[https://gitlab.com/luispscarvalho/datasets/-/tree/UFBA2020\\_Thesis/study\\_I](https://gitlab.com/luispscarvalho/datasets/-/tree/UFBA2020_Thesis/study_I)

We further describe the package's items:

1. The agreement dataset: it comprises a series of spreadsheets corresponding to each of our action research study's rounds of evaluation. As we counted on the help of two software development specialists, the dataset contains two spreadsheets per round, in the format that we presented in Section 4.3.1 (Table 4.2);
2. A copy of one of our papers: as we explained in Section 4.4.1, we wanted to take advantage of raters' know-how to refine our method. As a way to instruct them about it, we added a copy of one of our papers (CARVALHO; NOVAIS; MENDONÇA, 2018). We expected that the paper would enlightened the raters about our approach

---

<sup>3</sup><https://gitlab.com/luispscarvalho/datasets>

and provided them with the necessary knowledge regarding concerns identification and analyses, so they would be able to suggest corrections and improvements;

3. Scripts: in Section 3.1.2, we pointed out that we rely on scripts to automate our studies' selection and analysis of data. We added three scripts to the package: (i) "aks.R" encapsulates some generic R-language routines to filter, organize, split, join, agglomerate and visualize the concerns-related information kept in our datasets, (ii) "preparation.R" is capable of selecting random samples of artifacts to create the agreement dataset (the one that we sent to raters), and (iii) "evaluation.R" that processes raters' tagged spreadsheets through the Kappa Agreement Coefficient (described in Section 2.8)

To reuse this replication package, we recommend the workflow exhibited in Figure 6.2. First step comprises downloading the package from the aforementioned url (Activity 1) and executing AKS to mine concerns from software projects. Second step requires running the "aks.R" script to load a collection of routines that we developed to support the processing of concerns (Activity 2). Then, it is necessary to execute the "preparation.R" script to produce an agreement dataset (Activity 3). Raters must tag the dataset (Activity 4) before the execution of the "evaluation.R" script (Activity 5) that determines the strength of their agreement.

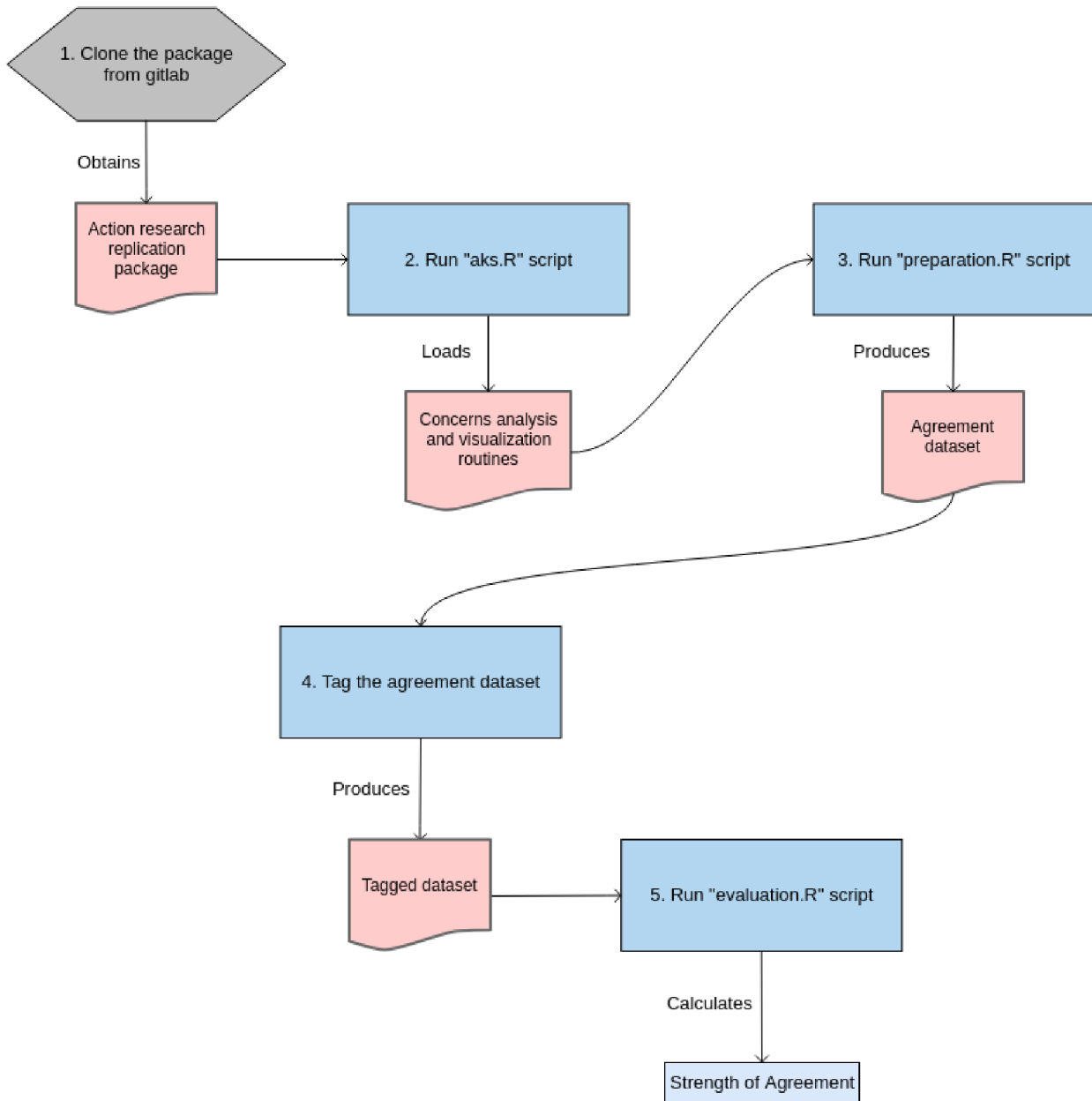
### 6.2.2 Study II and III's Replication Package

We decided to join Study II's (Chapter 5) and Study III's (Appendix C) replication packages as one because: (i) we used the same software projects, non-relational databases, to create both studies' datasets and (ii) Study II is influenced by Study III's finding and conclusions, *e.g.*, we applied the same mining strategies that we developed during Study III to extract the evolution of our metric, DtC, from the databases. The following is the location of the studies' replication package:

[https://gitlab.com/luispscavalho/datasets/-/tree/UFBA2020\\_Thesis/studies\\_II\\_and\\_III](https://gitlab.com/luispscavalho/datasets/-/tree/UFBA2020_Thesis/studies_II_and_III)

The package contains the following items:

1. Studies' dataset: the dataset is composed of two subsets, one that contains data about the evolution of DtC (the focus of Study I) and another one with data about source code complexity (the purpose of Study III). This means: we based both studies on the evolution of non-relational databases' concerns, but we specialized the subsets to address two distinct aspects related to systems' historical data.
2. Scripts: the package includes three scripts: (i) "aks.R" encloses our library of routines to support analysis and visualization of concerns, (ii) "complexity.R" is a specialized script that uses "aks.R" to investigate the evolution of source code complexity, and (iii) "dtc.R", which also specializes "aks.R" routines to process and visualize the evolution of DtC.

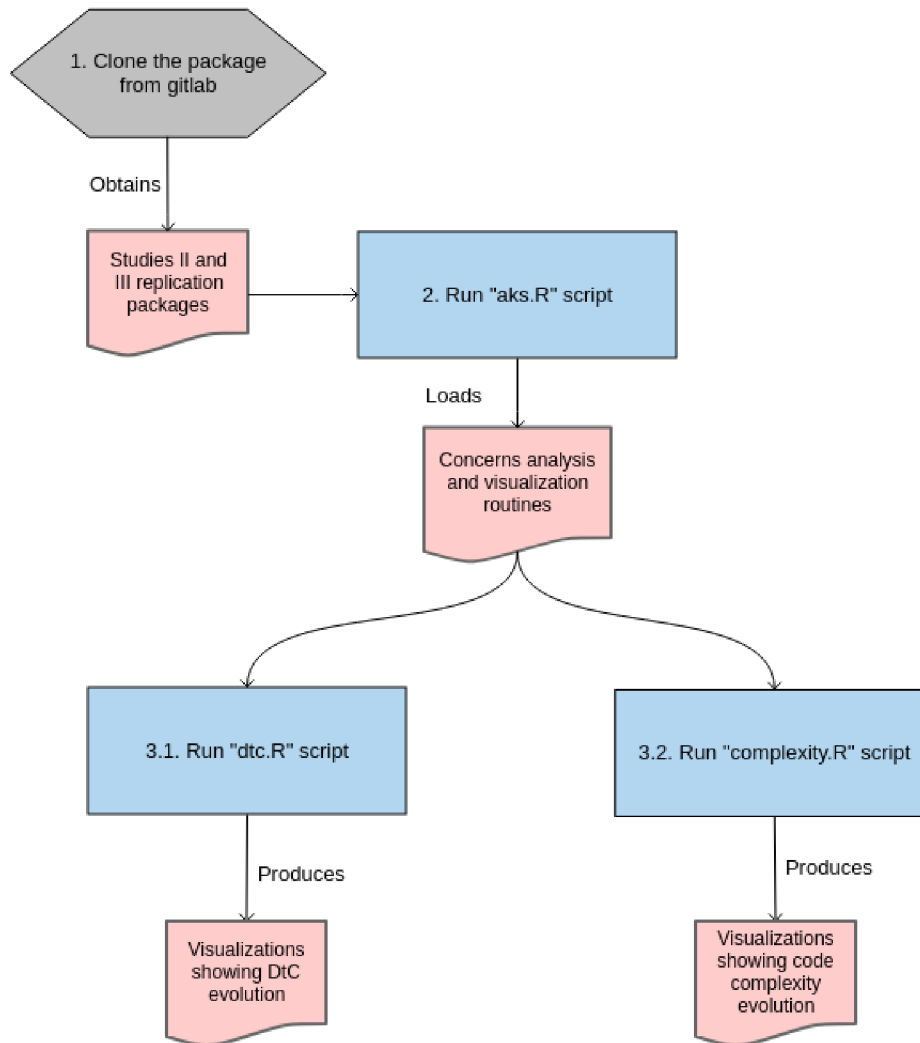


**Figure 6.2** Reuse of our Action Research Replication Package

We propose the workflow in Figure 6.3 as a guide to reuse the replication package. First step requires cloning the package from gitlab (Activity 1). As our analysis scripts depend on the generic routines that we placed in “aks.R”, it is necessary to execute it first (Activity 2). Then, it is possible to replicate either Study II by running the “dte.R” script (Activity 3.1) or Study III by executing the “complexity.R” script (Activity 3.2).

We placed the descriptions of Studies II’s and III’s datasets in Appendix G. Section G.1 shows the format of Study II’s spreadsheets. Section G.2 describes the columns of Study III’s spreadsheets.





**Figure 6.3** Reuse of our Studies II and III Replication Package

Study IV (in Appendix E) has the purpose of knowing if it is possible to apply our method to mine concerns from systems that are developed under different technological contexts. Its dataset contains information about concerns identified from javascript applications. As it is a very preliminary investigation, the dataset is not as precise and complete as previous studies' datasets. However, we believe that Study IV's replication package is a good starting point to gather a basic knowledge about how javascript developers implement concerns. The following package can be used to replicate the study:

[https://gitlab.com/luispcarvalho/datasets/-/tree/UFBA2020\\_Thesis/study-IV](https://gitlab.com/luispcarvalho/datasets/-/tree/UFBA2020_Thesis/study-IV)

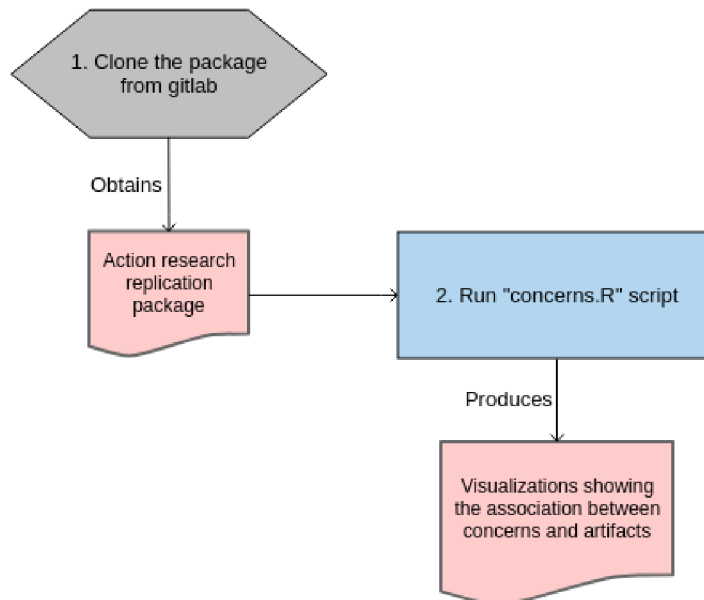
This content of the replication package comprises the following elements:

1. Study's dataset: it shows a simple association between concerns and source code artifacts mined from two domains of software: role-playing games and chats. As

we have not defined routines to calculate javascript systems' DtC yet, in Study IV we rely on the Lines of Code (LOC) metric to examine and visualize concerns. It is also important to point out that the dataset refers to one single snapshot of the software projects' source code;

2. Script: the package has only one script: "concerns.R" has all the necessary routines to replicate Study IV.

We suggest the workflow shown in Figure 6.4 to replicate Study IV. The replication package must be downloaded/cloned from its repository (Activity 1). Then, the "concerns.R" script must be executed to visualize the relationship between concerns and javascript applications (Activity 2).



**Figure 6.4** Reuse of our Study IV Replication Package

Section G.3 in Appendix G describes all columns of Study IV's spreadsheets.

### 6.3 OUR PUBLICATIONS

As another measure to advertise our method and promote the replication of our studies, we did our best to publish our findings. In Table 6.1, we mention the works that we published as a result of the studies described in this work. The table also describes other publications that are out of the scope of our research, but are related to some theoretical and technical concepts which we have made use of: software repositories mining, static processing of source code artifacts, software evolution and visualization. The *Contribution* column informs how the author of this thesis contributed for the publication: as a **Co-author** or **Main Author**. *Category* identifies the scope of the contribution: **Primary** (the publication is in the scope of this work) and **Secondary** (the publication is not in

the scope of this work). *Type* refers to the type of the publication: **Conference**, **Book Chapter**, **Workshop** and **Journal**.

We highlight that our paper, *Investigating the Relationships between Code Smell Agglomerations and Architectural Concerns: Similarities and Dissimilarities from Distributed, Service-Oriented and Mobile Systems* (CARVALHO; NOVAIS; MENDONÇA, 2018), was awarded as one of the best papers (second place) during 2018's Brazilian Symposium on Components, Architectures and Software Reuse (SBCARS).

## 6.4 CONCLUSION

We believe that our method and tool can become important assets in the area of concerns identification and evaluation of how they impact software development. Achieving this requires disseminating the findings of our studies, method, tool and datasets.

As a practical dimension of our actions regarding dissemination, we have made our datasets and the source code of our tool available for reuse via online repositories. We also licensed them under a permissive version of GLP (GLP3). Hopefully, this will attract researchers' and practitioners' attention and they will feel motivated to reuse our method and propose/contribute-with improvements.

We have also published our studies' results whenever we are given a chance, but we have not presented some content of this thesis to the academic community yet. We intend to publish the results of Studies I (Chapter 4) and II (Chapter 5) as soon as possible.

Table 6.1 Our Publications

Title	Contribution	Category	Type	Reference
A systematic mapping study on mining software repositories	Co-author	Primary	Conference	(FARIAS et al., 2016)
RepositoryMiner - uma ferramenta extensível de mineração de repositórios de software para identificação automática de Divida Técnica	Co-author	Primary	Workshop	(MENDES et al., 2017)
Investigating the Relationship between Code Smell Agglomerations and Architectural Concerns: Similarities and Dissimilarities from Distributed, Service-Oriented and Mobile Systems	Main Author	Primary	Conference	(CARVALHO; NOVAIS; MENDONÇA, 2018)
Relationships between Design Problem Agglomerations and Concerns Having Types and Domains of Software as Transverse Dimensions	Main Author	Primary	Journal	(CARVALHO; NOVAIS; MENDONÇA, 2020)
An Ontology-based Approach for Analyzing the Occurrence of Code Smells in Software	Main Author	Secondary	Conference	(CARVALHO; NOVAIS; MENDONÇA, 2017)
An Approach for Semantically-Enriched Recommendation of Refactorings Based on the Incidence of Code Smells	Main Author	Secondary	Book Chapter	(CARVALHO et al., 2017)
VISMELLS: An Interactive Visualization for Identifying and Evaluating the Effects of Code Smells on Software Projects	Co-author	Secondary	Conference	(SILVA et al., 2018)
VisMinerService A REST Web Service for Source Mining	Main Author	Secondary	Conference	(CARVALHO; NOVAIS; MENDONÇA, 2015)

*The purpose of computing is insight, not numbers – Richard Hamming*

## CONCLUSION

We described our effort in automating the mining and processing of concerns with the help of information that developers store in third-party components' metadata. We believe that we have been able to circumvent some limitations we initially identified: (i) the lack of and inadequacies of Software Requirements Documents (SRDs) and Software Architecture Documents (SADs) and available automatic approaches; and (ii) error-proneness of approaches that focus on the manual identification and analyses of concerns. As a result, we created and evaluated a new method.

Customarily, developers add metadata files to software projects as a way to automate the injection of components in the source code. As this has turned into a widespread good practice, projects that contain metadata about components became abundant and we saw this as an opportunity to locate and process information about concerns. Developers also often synchronize the metadata files with Version Control Systems (VCS). This ensures that they are updated frequently and evolve together with other source code artifacts. So, they can enable the investigation of historical data about how developers integrate components in systems' source code to implement concerns.

We counted on the help of software development specialists (Chapter 4) to enhance our method and tool. We regard our action research study as a valuable experience because it helped us to reduce the bias of our initial guesswork, during which we defined the layout of our method and the strategies behind the mining of concerns (Chapter 3). We met the specialists' opinions on how to identify concerns and measuring the Dedication to Concern (DtC) to a moderate degree. To the best of our knowledge, we took one step ahead in using static analysis of source code to emulate how developers perceive the relationship between software projects and concerns.

Spotting cases in which third-party components are scarcely used to implement concerns was one important limitation that we identified and managed to solve. We wrapped this problem within a broader concept: the evolution of DtC (Chapter 5). We found out that the implementation of some concerns tends to avoid the principle of separation of concerns. We believe that choosing between separating and combining concerns in source

code artifacts is subject to the use that developers can make of each third-party component. Hence, some cases in which developers do not promote the separation are not entirely unacceptable.

## 7.1 THE WAY WE FULFILLED OUR RESEARCH'S GOAL

Now, we summarize our conclusions regarding the fulfillment of our research's goal. As defined in Chapter 1, we have the following purpose: **taking advantage of third-party components' metadata as a way to extract and analyze information about the implementation of concerns through the historical data of systems**. We believe that we have achieved the goal by allowing:

1. The mining of several software projects: as many developers have adhered to the use of metadata files to store information about third-party components (*e.g.*, POM and Gradle files), this type of information has become plentiful. This allowed us to analyze concerns extracted from a multitude of software projects;
2. Keeping track of the evolution of third-party components' metadata: developers store the metadata files in VCS. This means: as we mined the historical data of software projects, we were able to recover information about the evolution of components. This enabled us to propagate the analysis of concerns through many versions of systems.

Apart from the aforementioned main achievements, we also highlight that we fulfilled the following:

1. Grouping systems under transverse dimensions: we sought to understand the effects of grouping software projects before analyzing concerns (Appendix C). Doing this enhanced our investigations because it allowed us to find concerns being shared by projects that have either the same context of use (domains of software) or target the same development platform (types of software);
2. Demonstrating that our method is capable of supporting the analysis of concerns under varied contexts of software development: we perceived resemblances in the way how java and javascript developers keep track of third-party components (Appendix E). Both rely on special files to indicate which components must be integrated into their systems: POM/Gradle files (java) and package.json files (javascript). As a result, we showcased different scenarios in which we were able to instantiate our method to deal with different development technologies.

## 7.2 FUTURE WORK

Future work includes improvements and new evaluations concerning the threats to validity and proposed enhancements that we discussed throughout the studies that we conducted (Chapters 4 and 5 – Appendixes C and E). In the next paragraphs, we highlight some of them.

In Study I (Chapter 4) and II (Chapter 5), we focused on identifying concerns and measuring one specific metric, DtC, and its evolution. However, as described in Section 3.1.2, our method is not particularly limited to calculate DtC. It can be adapted to include other metrics and to combine them with DtC. For instance, we must rerun Study III (Appendix C) to associate our investigations about source code complexity and transverse dimensions with DtC.

Learning about the patterns that developers follow when they have to implement concerns is another promising future study. Tracking the way how the “Security” concerns is added to the source code may reveal specific standards and strategies that software security experts tend to follow. Perhaps, such strategies differ from other ones adopted by developers when they need to implement, for instance, “Test” “Geospatial Processing”. In other words, we think that it is possible to use our method to gather insights about tendencies in the way how developers materialize different aspects of software systems. Then, we might be able to create a corpus of good usual practices to serve as a guide for future concerns implementations.

We unveiled a broader set of scenarios to apply our method by conducting Study IV (Appendix E). Injecting metadata in software projects to keep track of third-party components is a common practice and it is independent of programming technologies. We see this as another chance to advance our research. There is plenty to investigate on how to make our approach available to support developers that work with languages other than java, *e.g.*, javascript, python, PHP and ruby.

An important evolution of our research resides in varying the thresholds of our DtC measurement rule (Section 3.2). The thresholds that we embedded in AKS and used to produce our studies’ dataset were determined by guesswork after we examined some software projects’ source code artifacts. By the time we evaluated our method (in Chapter 3), we did not take advantage of the raters to fine-tune the thresholds. Our focus on applying raters’ comments to define and improve strategies to mine the source code deviated our attention from adjusting our DtC measurement rule. We must conduct new studies to observe how changing the rule’s thresholds values impacts our method and findings.

Our action research study (described in Chapter 4) provided us with many insights. For example, enhancing our method and tool by processing semantic information found in software projects is an important next step. We did our best to approximate the static analysis of source code to software specialists’ points of view. However, we believe that we can advance our approach by extracting conceptual/semantic information from elements that our method (and AKS) currently discard: keywords from the name of packages, classes, methods, the comments that developers add to the source code, and Software Architecture Documents (if available). We must also investigate if commits’ messages can become a source of semantically enriched data about concerns.

In Figure 7.1, we propose a sequence of studies toward the definition of a Unified Concerns Identification Heuristic (UCIH). The scope of this thesis is limited to the static processing of source code artifacts, *e.g.*, java files and components’ metadata. We conjecture that a new series of investigations about the semantic processing of systems’ source code is necessary to enhance our method and tool, and to complement our findings. We

believe that we can better reflect the way how developers manage concerns if we merge static and semantic analyses to create a heuristic. Basing a new action research study on the template proposed by Santos and Travassos (2011) is highly desirable to validate and refine the UCIH. It would also be interesting to generalize it to deal with other development technologies/contexts, *e.g.*, javascript, python, PHP, and ruby.

As a result of the interview that we described in Section 4.4.4, we collected some ideas for future practical applications of our method from raters. For example, although raters said that parameterizing the analysis of *interfaces* to either include or exclude them is a good decision, but they also said that this requires a deeper investigation. They even pointed out that processing interfaces' children classes would be a good indirect approach to measure the DtC of this type of class.

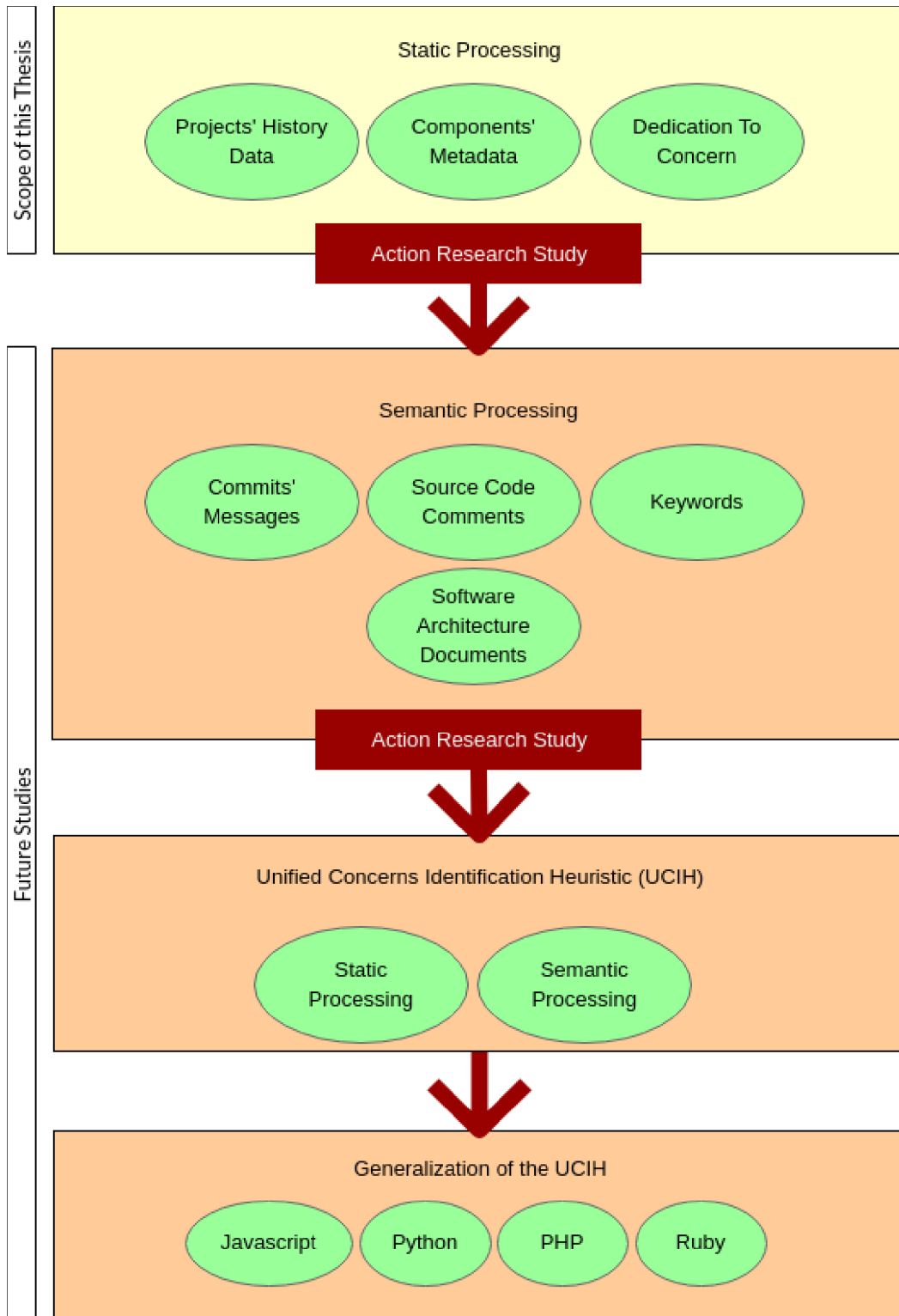
Raters suggested that DtC can be used to validate layered software architectures. For instance, layers of projects that follow the Model-View-Controller (MVC) pattern have distinct purposes (DEACON, 2009). It is undesirable to notice a moderate/high DtC of the "Database" concern in the "View" layer. Oppositely, concerns that deal with user interface features (*e.g.*, "UI", "Visualization") are welcomed in the "View". So, depending on how concerns and layers pair with each other, different DtC's thresholds should be expected. As an extension, they also proposed associating different levels of "warning" to the undesired presence of concerns in software layers.

According to one of the raters, it is not always possible to fully eliminate unexpected concerns from a particular layer. This comes from developers having to deal with tight schedules and aggressive deadlines. Consequently, they are often unable to refine software source code. Even though, they cannot release a new software version if some concerns appear in the wrong layer. The presence of the "Database" concern in the "View" layer fits in this category. The rater sees it as a security problem because it may enable hacking via SQL injection, potentially allowing attackers to obtain unrestricted access to sensitive information (HALFOND et al., 2006). This goes against one of the MVC's advantages (NYSTROM, 2007): applying processing rules to data received from users to ensure it is normalized and safe, which can guard applications from malicious inputs, such as SQL injections. On the other hand, developers may misplace other concerns in the "View", but they are still able to release a new version if the concerns are less impacting, *i.e.*, executing data "Validation" routines in the "View" while they should better happen in the "Control" and "Model" layers.

We believe that analyzing the relationship between concerns and layered software architecture (like MVC) demands incorporating Technical Debt (TD) theoretical/technical aspects into our method. As TD refers to delayed tasks that may require extra effort in the future (CUNNINGHAM, 1992)(FARIAS et al., 2020), releasing systems with misplaced concerns can be seen as developers postponing a necessary refactoring. In this context, according to raters' point of view, some concerns can lead to a higher debt than others. Delaying the removal of the "Database" concern from the "View" layer accumulates more debt than other less critical ones, as "Validation".

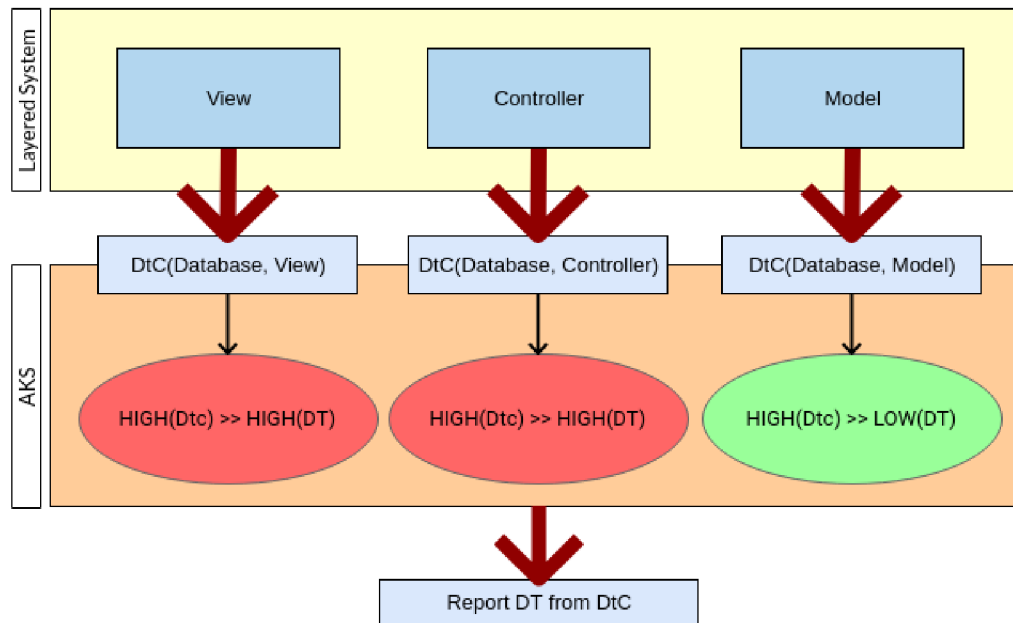
Figure 7.2 depicts DtC's potential to spot and calculate TD. Our DtC measurement rule (described in Section 3.1.4.1) would have to consider one extra parameter to process TD: layers of layered system. Finding a high DtC regarding the "Database" concern in





**Figure 7.1** Toward a Unified Concerns Identification Heuristic (UCIH)

artifacts of the “View” layer would cause AKS to report a high TD. On the other hand, detecting a high DtC from the artifacts that implement “Database” in the “Model” layer would indicate a low (or a nonexistent) TD.



**Figure 7.2** From DtC to Technical Debt

Applying DtC to support the detection/measurement of TD adds another challenge to the limitation that we aforementioned: determining an adequate set of thresholds for our metrics and DtC measurement rule. The thresholds would depend not only on the relationship between concerns and artifacts. We would have to tailor them to suit every single association between concerns and layers as well.

Raters see another opportunity in applying the evolution of DtC to manage the making of software. Phases of development can be planned in a way to consider the gradual appearance of concerns through time. “Security”, “Test”, “Database” are examples of concerns that may be expected to occur since early cycles of development and keep on growing until later phases. Scheduling the release of systems’ versions would be determined by concerns’ DtC reaching specific thresholds through time.

## BIBLIOGRAPHY

- ABILIO, R. et al. Detecting code smells in software product lines – an exploratory study. In: *2015 12th International Conference on Information Technology - New Generations*. [S.l.: s.n.], 2015. p. 433–438.
- ABILIO, R. et al. Metrics for feature-oriented programming. In: IEEE. *2016 IEEE/ACM 7th International Workshop on Emerging Trends in Software Metrics (WETSoM)*. [S.l.], 2016. p. 36–42.
- ADAMS, B.; JIANG, Z. M.; HASSAN, A. E. Identifying crosscutting concerns using historical code changes. In: ACM. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. [S.l.], 2010. p. 305–314.
- AGGARWAL, C. C. *Data mining: the textbook*. [S.l.]: Springer, 2015.
- AGÜERO, M.; BALLEJOS, L. Dependency management in the cloud: An efficient proposal for java. In: *2017 XLIII Latin American Computer Conference (CLEI)*. [S.l.: s.n.], 2017. p. 1–9.
- AINI, Q.; ZULIANA, S. R.; SANTOSO, N. P. L. Management measurement scale as a reference to determine interval in a variable. *Aptisi Transactions On Management, Asosiasi Perguruan Tinggi Swasta Indonesia*, v. 2, n. 1, p. 45–54, 2018.
- ALVAREZ, R. M.; KEY, E. M.; NÚÑEZ, L. Research replication: Practical considerations. *PS: Political Science & Politics*, Cambridge University Press, v. 51, n. 2, p. 422–426, 2018.
- ALVES, T. L.; YPMA, C.; VISSER, J. Deriving metric thresholds from benchmark data. In: IEEE. *2010 IEEE International Conference on Software Maintenance*. [S.l.], 2010. p. 1–10.
- BADAMPUDI, D.; WOHLIN, C.; PETERSEN, K. Software component decision-making: In-house, oss, cots or outsourcing—a systematic literature review. *Journal of Systems and Software*, Elsevier, v. 121, p. 105–124, 2016.
- BAETJER, H. *Software as capital: An economic perspective on software engineering*. [S.l.]: IEEE Computer Society Press, 1997.
- BAJAJ, K.; PATEL, H.; PATEL, J. Evolutionary software development using test driven approach. In: IEEE. *Computing and Communication (IEMCON), 2015 International Conference and Workshop on*. [S.l.], 2015. p. 1–6.

- Baojiang Cui et al. Code comparison system based on abstract syntax tree. In: *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*. [S.l.: s.n.], 2010. p. 668–673.
- BARROS-JUSTO, J. L. et al. What software reuse benefits have been transferred to the industry? a systematic mapping study. *Information and Software Technology*, Elsevier, v. 103, p. 1–21, 2018.
- BARRY, E. J.; KEMERER, C. F.; SLAUGHTER, S. A. On the uniformity of software evolution patterns. In: IEEE COMPUTER SOCIETY. *Proceedings of the 25th International Conference on Software Engineering*. [S.l.], 2003. p. 106–113.
- BASKERVILLE, R.; MYERS, M. D. Special issue on action research in information systems: Making is research relevant to practice: Foreword. *MIS quarterly*, JSTOR, p. 329–335, 2004.
- BASKERVILLE, R. L. Investigating information systems with action research. *Communications of the association for information systems*, v. 2, n.1, p. 19, 1999.
- BELLOMO, S. et al. Evolutionary improvements of cross-cutting concerns: Performance in practice. In: IEEE. *2014 IEEE International Conference on Software Maintenance and Evolution*. [S.l.], 2014. p. 545–548.
- BERNARDI, M. L.; CIMITILE, M.; LUCCA, G. D. Mining static and dynamic crosscutting concerns: a role-based approach. *Journal of Software: Evolution and Process*, Wiley Online Library, v. 28, n. 5, p. 306–339, 2016.
- BERRY, K. J.; JR, P. W. M. A generalization of cohen’s kappa agreement measure to interval measurement and multiple raters. *Educational and Psychological Measurement*, Sage Publications Sage CA: Thousand Oaks, CA, v. 48, n. 4, p. 921–933, 1988.
- BÖHME, R.; FREILING, F. C. On metrics and measurements. In: *Dependability metrics*. [S.l.]: Springer, 2008. p. 7–13.
- BORGES, H. et al. On the popularity of github applications: A preliminary note. *arXiv preprint arXiv:1507.00604*, 2015.
- BOWES, D. et al. How good are my tests? In: *2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM)*. [S.l.: s.n.], 2017. p. 9–14. ISSN 2327-0969.
- BRENNAN, R. L.; PREDIGER, D. J. Coefficient kappa: Some uses, misuses, and alternatives. *Educational and psychological measurement*, Sage Publications Sage CA: Thousand Oaks, CA, v. 41, n. 3, p. 687–699, 1981.
- BRUNTINK, M. et al. An evaluation of clone detection techniques for crosscutting concerns. In: IEEE. *20th IEEE International Conference on Software Maintenance, 2004. Proceedings*. [S.l.], 2004. p. 200–209.

- BUSCH, A. et al. Assessing the quality impact of features in component-based software architectures. In: SPRINGER. *European Conference on Software Architecture*. [S.l.], 2019. p. 211–219.
- CAMPBELL, D. T.; STANLEY, J. C. *Experimental and quasi-experimental designs for research*. [S.l.]: Ravenio Books, 2015.
- CANFORA, G.; CERULO, L. How crosscutting concerns evolve in jhotdraw. In: IEEE. *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*. [S.l.], 2005. p. 65–73.
- CARVALHO, L. P.; NOVAIS, R.; MENDONÇA, M. Investigating the relationship between code smell agglomerations and architectural concerns: Similarities and dissimilarities from distributed, service-oriented, and mobile systems. In: *2018 XII Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. [S.l.: s.n.], 2018.
- CARVALHO, L. P. da S.; NOVAIS, R. L.; MENDONÇA, M. Relationships between design problem agglomerations and concerns having types and domains of software as transverse dimensions. *Journal of the Brazilian Computer Society*, v. 26, n. 1, p. 5, Jul 2020. ISSN 1678-4804. Disponível em: <https://doi.org/10.1186/s13173-020-00099-y>.
- CARVALHO, L. P. da S. et al. An approach for semantically-enriched recommendation of refactorings based on the incidence of code smells. In: SPRINGER. *International Conference on Enterprise Information Systems*. [S.l.], 2017. p. 313–335.
- COHEN, J. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, Sage Publications Sage CA: Thousand Oaks, CA, v. 20, n. 1, p. 37–46, 1960.
- CONGER, A. J. Integration and generalization of kappas for multiple raters. *Psychological Bulletin*, American Psychological Association, v. 88, n. 2, p. 322, 1980.
- CUNNINGHAM, W. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, ACM New York, NY, USA, v. 4, n. 2, p. 29–30, 1992.
- DAVISON, R.; MARTINSONS, M. G.; KOCK, N. Principles of canonical action research. *Information systems journal*, Wiley Online Library, v. 14, n. 1, p. 65–86, 2004.
- DAWSON, R.; O'NEILL, B. Simple metrics for improving software process performance and capability: a case study. *Software Quality Journal*, Springer, v. 11, n. 3, p. 243–258, 2003.
- DEACON, J. Model-view-controller (mvc) architecture. *Online* [Citado em: 10 de março de 2006.] <http://www.jdl.co.uk/briefings/MVC.pdf>, 2009.
- DELEV, T.; GJORGJEVIKJ, D. Static analysis of source code written by novice programmers. In: IEEE. *2017 IEEE Global Engineering Education Conference (EDUCON)*. [S.l.], 2017. p. 825–830.

DIAS, R. S. et al. Effects of visualizing technical debts on a software maintenance project. In: *Proceedings of the XVIII Brazilian Symposium on Software Quality*. [S.l.: s.n.], 2019. p. 39–48.

DÍAZ-PACE, J. A. et al. Producing just enough documentation: An optimization approach applied to the software architecture domain. *Journal on Data Semantics*, Springer, p. 37–53, 2016.

DIT, B. et al. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process*, Wiley Online Library, v. 25, n. 1, p. 53–95, 2013.

DONKER, D.; HASMAN, A.; GEIJN, H. V. Interpretation of low kappa values. *International journal of bio-medical computing*, Elsevier, v. 33, n. 1, p. 55–64, 1993.

DÓSEA, M.; SANT’ANNA, C.; SILVA, B. C. da. How do design decisions affect the distribution of software metrics? 2018.

EADDY, M. et al. Do crosscutting concerns cause defects? *IEEE transactions on Software Engineering*, IEEE, v. 34, n. 4, p. 497–515, 2008.

ELLIS, J.; BELLE, J.-P. V. Open source software adoption by south african mses: barriers and enablers. In: *Proceedings of the 2009 Annual Conference of the Southern African Computer Lecturers’ Association*. [S.l.: s.n.], 2009. p. 41–49.

EMANUELSSON, P.; NILSSON, U. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, Elsevier, v. 217, p. 5–21, 2008.

FARIAS, M. A. de F. et al. Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Information and Software Technology*, Elsevier, v. 121, p. 106270, 2020.

FARIAS, M. A. de F. et al. A systematic mapping study on mining software repositories. In: ACM. *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. [S.l.], 2016. p. 1472–1479.

FENTON, N. E.; NEIL, M. Software metrics: roadmap. In: *Proceedings of the Conference on the Future of Software Engineering*. [S.l.: s.n.], 2000. p. 357–370.

FONTANA, F. A. et al. Investigating the impact of code smells on system’s quality: An empirical study on systems of different application domains. In: *2013 IEEE International Conference on Software Maintenance*. [S.l.: s.n.], 2013. p. 260–269. ISSN 1063-6773.

FOWLER, M.; BECK, K. *Refactoring: improving the design of existing code*. [S.l.: s.n.], 1999.

FRANCESE, R.; RISI, M.; SCANNIELLO, G. Enhancing software visualization with information retrieval. In: *2015 19th International Conference on Information Visualisation*. [S.l.: s.n.], 2015. p. 189–194. ISSN 1550-6037.

- FRANCO-BEDOYA, O. et al. Open source software ecosystems: A systematic mapping. *Information and software technology*, Elsevier, v. 91, p. 160–185, 2017.
- GANDHI, N.; GONDWAL, N.; TANDON, A. Reliability modeling of oss systems based on innovation-diffusion theory and imperfect debugging. In: *ICITKM*. [S.l.: s.n.], 2017. p. 53–58.
- GÎRBA, T.; DUCASSE, S. Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, Wiley Online Library, v. 18, n. 3, p. 207–236, 2006.
- GOMES, F. et al. Uma análise da relação entre code smells e dívida técnica auto-admitida. In: . [S.l.: s.n.], 2019. p. 37–44.
- GÓMEZ, O. S.; JURISTO, N.; VEGAS, S. Understanding replication of experiments in software engineering: A classification. *Information and Software Technology*, Elsevier, v. 56, n. 8, p. 1033–1048, 2014.
- GOULAO, M. et al. Software evolution prediction using seasonal time analysis: A comparative study. In: *2012 16th European Conference on Software Maintenance and Reengineering*. [S.l.: s.n.], 2012. p. 213–222. ISSN 1534-5351.
- HALFOND, W. G. et al. A classification of sql-injection attacks and countermeasures. In: IEEE. *Proceedings of the IEEE international symposium on secure software engineering*. [S.l.], 2006. v. 1, p. 13–15.
- HAND, D. J.; ADAMS, N. M. Data mining. *Wiley StatsRef: Statistics Reference Online*, Wiley Online Library, p. 1–7, 2014.
- HANNEMANN, J.; KICZALES, G. Overcoming the prevalent decomposition of legacy code. In: *Workshop on Advanced Separation of Concerns*. [S.l.: s.n.], 2001. v. 167.
- HASSAN, A. E. The road ahead for mining software repositories. In: IEEE. *2008 Frontiers of Software Maintenance*. [S.l.], 2008. p. 48–57.
- HE, C.; YE, S. A method for identification of crosscutting concerns based on goal model and two-state algorithm. In: IEEE. *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*. [S.l.], 2015. v. 1, p. 431–435.
- HEMETSBERGER, A.; REINHARDT, C. Sharing and creating knowledge in open-source communities: The case of kde. In: *Paper for Fifth European Conference on Organizational Knowledge, Learning, and Capabilities, Innsbruck*. [S.l.: s.n.], 2004.
- HERNÁNDEZ, L.; COSTA, H. Identifying similarity of software in apache ecosystem – an exploratory study. In: *2015 12th International Conference on Information Technology - New Generations*. [S.l.: s.n.], 2015. p. 397–402.

- IBIAPINA, I. et al. Tdvision: Um módulo computacional para visualização de dívidas técnicas. In: SBC. *Anais da IV Escola Regional de Informática do Piauí*. [S.l.], 2018. p. 103–108.
- JUHÁR, J.; VOKOROKOS, L. Separation of concerns and concern granularity in source code. In: IEEE. *2015 IEEE 13th International Scientific Conference on Informatics*. [S.l.], 2015. p. 139–144.
- KAVALER, D. et al. Tool choice matters: Javascript quality assurance tools and usage outcomes in github projects. In: IEEE PRESS. *Proceedings of the 41st International Conference on Software Engineering*. [S.l.], 2019. p. 476–487.
- KAZMAN, R. et al. A case study in locating the architectural roots of technical debt. In: IEEE. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. [S.l.], 2015. v. 2, p. 179–188.
- KHOMYAKOV, I. et al. An analysis of automated technical debt measurement. In: SPRINGER. *International Conference on Enterprise Information Systems*. [S.l.], 2019. p. 250–273.
- KICZALES, G. Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 28, n. 4es, p. 154–es, 1996.
- KITCHENHAM, B.; PFLEEGER, S. L. Principles of survey research: part 5: populations and samples. *ACM SIGSOFT Software Engineering Notes*, ACM New York, NY, USA, v. 27, n. 5, p. 17–20, 2002.
- KOCH, S. Profiling an open source project ecology and its programmers. *Electronic Markets*, Routledge, v. 14, n. 2, p. 77–88, 2004.
- LANDIS, J. R.; KOCH, G. G. The measurement of observer agreement for categorical data. *biometrics*, JSTOR, p. 159–174, 1977.
- LANZA, M.; MARINESCU, R. *Object-Oriented Metrics in Practice*. [S.l.]: Springer Publishing Company, Incorporated, 2010.
- LAWRENCE, R. Research dissemination: actively bringing the research and policy worlds together. *Evidence & Policy: A Journal of Research, Debate and Practice*, Policy Press, v. 2, n. 3, p. 373–384, 2006.
- LE, D. M. et al. Relating architectural decay and sustainability of software systems. In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. [S.l.: s.n.], 2016. p. 178–181.
- LOMAS, J. Diffusion, dissemination, and implementation: who should do what? *Annals of the New York Academy of Sciences*, v. 703, p. 226–35, 1993.
- LONGHURST, R. Semi-structured interviews and focus groups. *Key methods in geography*, v. 3, n. 2, p. 143–156, 2003.



- MA, W. et al. Do we have a chance to fix bugs when refactoring code smells? In: IEEE. *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*. [S.l.], 2016. p. 24–29.
- MAGNAVITA, R. C.; NOVAIS, R. L.; MENDONÇA, M. G. Using evowave to analyze software evolution. In: *ICEIS (2)*. [S.l.: s.n.], 2015. p. 126–136.
- MARÇAL, I. et al. Techniques for the identification of crosscutting concerns: A systematic literature review. Springer, p. 569–579, 2016.
- MARCUS, A.; POSHYVANYK, D.; FERENC, R. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, IEEE, v. 34, n. 2, p. 287–300, 2008.
- MENDES, T. et al. Repositoryminer - uma ferramenta extensível de mineração de repositórios de software para identificação automática de dívida técnica. In: *CBSOFT 2017 - Sessão de Ferramentas*. [S.l.: s.n.], 2017.
- MENDES, T. S. et al. Visminertd: a tool for automatic identification and interactive monitoring of the evolution of technical debt items. *Journal of the Brazilian Computer Society*, Springer, v. 25, n. 1, p. 2, 2019.
- MENDES, T. S. et al. Visminertd: Uma ferramenta para identificação automática e monitoramento interativo de dívida técnica. 2015.
- MEYER, B. *Object-oriented software construction*. [S.l.]: Prentice hall New York, 1988.
- MO, R. et al. Hotspot patterns: The formal definition and automatic detection of architecture smells. In: IEEE. *2015 12th Working IEEE/IFIP Conference on Software Architecture*. [S.l.], 2015. p. 51–60.
- MOCKUS, A.; VOTTA, L. G. Identifying reasons for software changes using historic databases. In: *icsm*. [S.l.: s.n.], 2000. p. 120–130.
- MUNOZ, F. et al. Inquiring the usage of aspect-oriented programming: An empirical study. In: *2009 IEEE International Conference on Software Maintenance*. [S.l.: s.n.], 2009. p. 137–146. ISSN 1063-6773.
- MUNOZ, S. R.; BANGDIWALA, S. I. Interpretation of kappa and b statistics measures of agreement. *Journal of Applied Statistics*, Taylor & Francis, v. 24, n. 1, p. 105–112, 1997.
- NAYEBI, M. et al. Anatomy of functionality deletion. In: *Proceedings of the Conference on Mining Software Repositories (MSR'18), Gothenburg, Sweden*. [S.l.: s.n.], 2018.
- NOVAIS, R.; SANTOS, J. A.; MENDONÇA, M. Experimentally assessing the combination of multiple visualization strategies for software evolution analysis. *Journal of Systems and Software*, Elsevier, v. 128, p. 56–71, 2017.

NUÑEZ-VARELA, A. S. et al. Finding core crosscutting concerns from object oriented systems using information retrieval. In: *2017 5th International Conference in Software Engineering Research and Innovation (CONISOFT)*. [S.l.: s.n.], 2017. p. 18–24.

NUÑEZ-VARELA, A. S. et al. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, Elsevier, v. 128, p. 164–197, 2017.

NYSTROM, M. *SQL injection defenses*. [S.l.]: " O'Reilly Media, Inc.", 2007.

O'BRIEN, M. P.; BUCKLEY, J.; SHAFT, T. M. Expectation-based, inference-based, and bottom-up software comprehension. *Journal of Software Maintenance and Evolution: Research and Practice*, Wiley Online Library, v. 16, n. 6, p. 427–447, 2004.

OIZUMI, W. et al. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In: *Proceedings of the 38th International Conference on Software Engineering*. [S.l.: s.n.], 2016. (ICSE '16), p. 440–451.

OIZUMI, W. N. et al. When code-anomaly agglomerations represent architectural problems? an exploratory study. In: *2014 Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2014. p. 91–100.

OIZUMI, W. N. et al. On the relationship of code-anomaly agglomerations and architectural problems. *Journal of Software Engineering Research and Development*, 2015.

OLIVEIRA, P.; VALENTE, M. T.; LIMA, F. P. Extracting relative thresholds for source code metrics. In: IEEE. *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. [S.l.], 2014. p. 254–263.

OZA, N. V.; HALL, T. Difficulties in managing offshore software outsourcing relationships: An empirical analysis of 18 high maturity indian software companies. *Journal of Information Technology Case and Application Research*, Taylor & Francis, v. 7, n. 3, p. 25–41, 2005.

O'HAGAN, A. O.; COLEMAN, G.; O'CONNOR, R. V. Software development processes for games: a systematic literature review. In: SPRINGER. *European Conference on Software Process Improvement*. [S.l.], 2014. p. 182–193.

PALYART, M.; MURPHY, G. C.; MASRANI, V. A study of social interactions in open source component use. *IEEE Transactions on Software Engineering*, 2017.

PORUBÄN, J.; NOSÁL, M. Leveraging program comprehension with concern-oriented source code projections. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FUER INFORMATIK. *3rd Symposium on Languages, Applications and Technologies*. [S.l.], 2014.

PRESSMAN, R. S. *Software engineering: a practitioner's approach*. [S.l.]: Palgrave Macmillan, 2005.

- PUHAKAINEN, P.; SIPONEN, M. Improving employees' compliance through information systems security training: an action research study. *MIS quarterly*, JSTOR, p. 757–778, 2010.
- RIEHLE, D. The economic motivation of open source software: Stakeholder perspectives. *Computer*, IEEE, v. 40, n. 4, p. 25–32, 2007.
- ROBILLARD, M. P. et al. On-demand developer documentation. In: IEEE. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2017. p. 479–483.
- ROBILLARD, M. R.; MURPHY, G. C. Concern graphs: finding and describing concerns using structural program dependencies. In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. [S.l.: s.n.], 2002. p. 406–416.
- ROSENHAINER, L. Identifying crosscutting concerns in requirements specifications. In: CITESEER. *Proceedings of OOPSLA Early Aspects*. [S.l.], 2004.
- RUNESON, P. A survey of unit testing practices. *IEEE software*, IEEE, v. 23, n. 4, p. 22–29, 2006.
- SAIKIA, L.; SINGH, S. Feature extraction and performance measure of requirement engineering (re) document using text classification technique. In: IEEE. *2018 4th International Conference on Recent Advances in Information Technology (RAIT)*. [S.l.], 2018. p. 1–6.
- SANTOS, P. S. M. D.; TRAVASSOS, G. H. Action research can swing the balance in experimental software engineering. In: *Advances in computers*. [S.l.]: Elsevier, 2011. v. 83, p. 205–276.
- SANT'ANNA, C. et al. On the modularity assessment of software architectures: Do my architectural concerns count? In: *Proc. International Workshop on Aspects in Architecture Descriptions (AARCH. 07), AOSD*. [S.l.: s.n.], 2007. v. 7.
- SANT'ANNA, C. et al. On the reuse and maintenance of aspect-oriented software: An assessment framework. In: *Proceedings XVII Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2003. v. 26.
- SHAIKH, M.; LEE, C.-G. Aspect oriented re-engineering of legacy software using crosscutting concern characterization and significant code smells detection. *International Journal of Software Engineering and Knowledge Engineering*, World Scientific, v. 26, n. 03, p. 513–536, 2016.
- SHATNAWI, A. et al. Reverse engineering reusable software components from object-oriented apis. *Journal of Systems and Software*, p. 442–460, 2017.
- SILVA, B. C. da et al. Refactoring of crosscutting concerns with metaphor-based heuristics. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 233, p. 105–125, 2009.

SILVA, I. de J. et al. Vismells: An interactive visualization for identifying and evaluating the effects of code smells on software projects. In: IEEE. *2018 XLIV Latin American Computer Conference (CLEI)*. [S.l.], 2018. p. 40–49.

SOMMERVILLE, I. *Software Engineering*. 9th. ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0137035152, 9780137035151.

STARON, M. Reporting action research studies. In: *Action Research in Software Engineering*. [S.l.]: Springer, 2020. p. 191–213.

SUSMAN, G. I.; EVERED, R. D. An assessment of the scientific merits of action research. *Administrative science quarterly*, JSTOR, p. 582–603, 1978.

TRIFU, A.; MARINESCU, R. Diagnosing design problems in object oriented systems. In: IEEE. *12th Working Conference on Reverse Engineering (WCRE'05)*. [S.l.], 2005.

TSAI, J. For better or worse: Introducing the gnu general public license version 3. *Berkeley Tech. LJ*, HeinOnline, v. 23, p. 547, 2008.

TUFANO, M. et al. When and why your code starts to smell bad. In: IEEE PRESS. *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. [S.l.], 2015. p. 403–414.

VALE, T. et al. Twenty-eight years of component-based software engineering. *Journal of Systems and Software*, Elsevier, v. 111, p. 128–148, 2016.

VELÁZQUEZ-RODRÍGUEZ, C.; ROOVER, C. D. Automatic library categorization. In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. [S.l.: s.n.], 2020. p. 733–734.

VELIOGLU, S.; SELCUK, Y. E. An automated code smell and anti-pattern detection approach. In: *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*. [S.l.: s.n.], 2017. p. 271–275.

VIDAL, S. et al. Identifying architectural problems through prioritization of code smells. In: *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SB-CARS)*. [S.l.: s.n.], 2016. p. 41–50.

YU, Z. et al. Characterizing the usage and impact of java annotations over 1000+ projects. *arXiv preprint arXiv:1805.01965*, 2018.

## CONCERNS IDENTIFIED DURING STUDY I

Concern	Purpose
Service-Orientation	allows communication with (web)services
Compression	supports data compression
Configuration	performs automatic configuration of systems' modules and functionalities
Metrics and Measurement	measures systems' metrical/quality attributes
Web App Support	enables embedding of server-client protocols
Graph Computing	supports mathematical processing of graphs
Stream Processing	enables data streaming
Caching	supports definition of caching strategies
Text Processing	processes data in form of text
Encryption	supports data encryption
Directory Management	supports directory management, <i>e.g.</i> , LDAP
Geospatial Processing	supports processing of geospatial data
Test	automates self-testing of programs
Data Processing	processes basic data formats, <i>e.g.</i> , date, string
I/O Processing	processes information obtained from I/O devices
Logging	allows logging of programs' routines
Data Format Processing	imports-exports to-from data formats, <i>e.g.</i> , xml
Process Execution	executes external processes, <i>e.g.</i> , OS programs
Bytecode Processing	processes bytecode <i>e.g.</i> , from ASM compilers
Report	enables previewing and printing of reports
Database	enables communication between client applications and databases

<b>Concern</b>	<b>Purpose</b>
Serialization	supports serialization of data
Benchmark	enables benchmark tests in applications
Distributed Computing	adds distributed-computing features to software systems
ElasticSearch Processing	supports processing of document-based information
Visualization	allows visualization of data
Java Parsing/Compiling	enable parsing and compiling of java source code
Ruby Parsing/Compiling	enable parsing and compiling of ruby source code
Scala Parsing/Compiling	makes parsing and compiling of scala source code
Security	adds features to automate security, <i>e.g.</i> , encryption
Dependency Injection	makes injection of components during runtime possible (hotplug)
Command Line Parsing	makes parsing of command line strings possible
Monitoring	enables monitoring of a program's execution
Caching	supports definition of caching strategies
Tracing	allows processing of tracing stacks
Documentation	adds support for source code documentation
Authentication	enables authentication of users
Cloud Computing	allows communication with cloud services
Mailing	enables sending/receiving of electronic messages
Messaging	supports implementation of message queues
Validation	automates validation of data structures <i>e.g.</i> , j-beans validation
Programming Utilities	provides special data structures, <i>e.g.</i> , list, map, set
Sample/Example	source code samples to exemplify use of functionalities
Behaviour-Driven Programming	supports adoption of BDP principles
Transaction	enables management of transactions
Source Code Versioning	allows versioning of source code artifacts
Mathematical Processing	provides support for complex mathematical calculations

## CONCERNS IDENTIFIED DURING STUDY II

Concern	Purpose	Found in...(*)					
		J	N	T	H	K	O
Service-Orientation	allows communication with services or enables a system to provide services	X	X	X	X	X	
Compression	supports data compression	X		X	X	X	
Configuration	automatic configuration of systems' modules and functionalities	X	X	X	X		
Web App Support	enables embedding of service-client protocols	X	X	X	X	X	X
Graph Computing	supports mathematical processing of graphs	X		X			X
Text Processing	processes data in the form of text	X	X	X	X		
Geospatial Processing	processes geospatial data	X		X			
Test	automates self-tests of programs	X	X	X	X	X	X
I/O Processing	accesses/processes information obtained from I/O devices	X	X	X	X	X	
Logging	allows logging of routines executed by programs	X	X	X	X	X	X
Data Format Processing	imports-exports from-to data formats, <i>e.g.</i> , xml, json	X	X	X	X	X	X
Process Execution	executes external processes, <i>e.g.</i> , OS programs	X	X	X	X	X	X
Database	enables communication between client applications and databases	X	X	X	X	X	X

(\*) J – JanusGraph, N – Neo4j, T – Titan, H – Heroic, K – KairosDb, O – OpenTSDB

Concern	Purpose	Found in...(*)					
		J	N	T	H	K	O
Metrics and Measurement	measures systems' metrical/quality attributes	X	X	X	X	X	
Stream Processing	enables data streaming	X					
Serialization	supports serialization of data	X	X	X	X		X
Benchmark	enables benchmark tests	X		X			
Distributed Computing	adds distributed-computing features software systems	X		X			X
ElasticSearch Processing	supports processing of document-based information	X		X	X		
Data Processing	enables processing of dataset formats, <i>e.g.</i> , CSV	X	X	X	X	X	
Security	adds features related to security, <i>e.g.</i> , encryption		X				
Dependency Injection	makes injection of components during runtime possible (hotplug)		X		X	X	
Tracing	allows processing of tracing stacks				X		
Documentation	supports documentation of source code artifacts		X				
Command-line Parsing	automates interpretation of command lines	X	X	X	X	X	
Authentication	enables authentication of users	X				X	
Cloud Computing	allows communication with cloud services	X			X		
Mailing	enables sending/receiving of electronic messages				X		
Messaging	implements message queues				X		
Validation	automates validation of data structures during runtime, <i>e.g.</i> , j-beans validation	X				X	
Programming Utilities	provides special data structures, <i>e.g.</i> , lists	X	X	X	X	X	X
Bytecode Processing	allows interpretation of bytecode, <i>e.g.</i> , java bytecode	X		X			
Samples/Examples	lessons about how to use APIs and systems		X				
Behavioural-Driven	enables adoption of BDD		X				
Transaction	allows management of transactions		X				
Mathematical Processing	provides support for complex mathematical calculations	X	X	X			

(\*) J – JanusGraph, N – Neo4j, T – Titan, H – Heroic, K – KairosDb, O – OpenTSDB



Concern	Purpose	Found in...(*)					
		J	N	T	H	K	O
Visualization	enables visualization of data	X	X	X			
Caching	supports adoption of caching strategies	X		X			
Encryption	enables data encryption	X					
Source Code Versioning	allows integration with VCS				X		
Java Parsing/Compiling	enables parsing and compiling of java programs	X					
Ruby Parsing/Compiling	enables parsing and compiling of ruby programs	X					
Scala Parsing/Compiling	enables parsing and compiling of scala programs	X					
Directory Management	supports processing of directory structures, <i>e.g.</i> , LDAP	X					
Web Server Support	allows embedding of web servers	X		X	X		
Monitoring	enables monitoring of programs' execution	X				X	

(\*) J – JanusGraph, N – Neo4j, T – Titan, H – Heroic, K – KairosDb, O – OpenTSDB



## STUDY III – ANALYZING TYPES AND DOMAINS OF SOFTWARE AS TRANSVERSE DIMENSIONS

Developers usually attempt to guarantee the quality of software projects as they evolve. For instance, they may try to find and manage the occurrence of design problems in the making of systems (VIDAL et al., 2016)(LE et al., 2016). Trifu and Marinescu (2005) define design problems as flaws in the making of software projects that impact their maintainability negatively. For instance, code smells (or smells) are potential indication of problems in the source code of information systems (FOWLER; BECK, 1999). Therefore, it is important to define strategies to spot and mitigate them whenever possible.

An example of strategy can be found in the work of Oizumi *et al.* (2016): they noticed that when smells interconnect as agglomerations they show more potential to compromise the quality of systems. Now, we examine the relationship between the code complexity caused by code smells agglomerations and concerns. IEEE defines code complexity<sup>1</sup> as “the degree to which a system or component has a design or implementation that is difficult to understand and verify”. To the best of our knowledge, the association between smells-related complexity and concerns obtained from third-party components’ metadata is a valuable investigation that has not been addressed yet.

Additionally, we believe that analyzing the association between complexity and concerns can be influenced to the way how we group software projects. In other words, we conjecture that it is important to define strategies to arrange software projects together as a way to optimize the mining of concerns. The strategies can rely on similarities between the concerns embedded into projects that are of the same type/domain. For instance, the following rationale exemplifies the applicability of types of software as a method to group systems to evaluate their complexity:

1. Different types of software may require the implementation of common concerns. For instance, regardless the type of software, developers may always add “Test” routines to validate systems’ functionalities;

---

<sup>1</sup><https://standards.ieee.org/standard/610-1990.html>

2. In opposition, other concerns remain particular to specific types of software. User Interface (“UI”) is not typically implemented by web services, but it is a feature generally found in other types of software, *e.g.*, mobile and web applications;
3. If a certain design problem affects concerns that are common to different software projects, it may be advantageous to define global strategies to mitigate the problem. For example, complexity increases in source code artifacts that host code smells (LANZA; MARINESCU, 2010)(FONTANA et al., 2013)(VELIOGLU; SELCUK, 2017). If “Test” is the main concern implemented by many smelly artifacts of systems “M” (a mobile system), “S” (a service-oriented system), and “D” (a distributed system), developers may find it interesting to define a global/general strategy to jointly remove smells from the test-related artifacts of “M”, “S”, and “D”. This can reduce their complexity and developers would have a chance to unify each system’s design problems mitigation approaches regardless their distinct types.

With this study, we seek to examine the association between concerns and agglomerations of code complexity caused by smells. Another aspect of our investigation: we want to observe how grouping software projects according to their types (distributed, service-oriented, and mobile systems) and domains (graph and time series databases) impact our analyses.

We compare the strategies (grouping systems regarding their types X grouping systems regarding their domains) by evaluating them according to two criteria: (i) the number of concerns shared by the grouped projects and (ii) if/how each group produces agglomerations of code complexity that are uniform through time. Evaluating the uniformity of agglomerations comes from perceiving that some systems are modified and stay productive for many years, while others are soon replaced or discontinued. Some systems suffer few changes and others undergo constant changes. As a consequence, it is possible to affirm that software projects do not follow uniform patterns during their lifecycle (BARRY; KEMERER; SLAUGHTER, 2003)(GOULAO et al., 2012). This can impact the use of concerns to agglomerate and analyze design problems. For instance, perhaps not all versions of the software projects agglomerate instances of code complexity. As well, it may not be the case that the agglomerations follow a successive, uniformly spaced pattern through time. If so, our method is less likely to help developers to define approaches to manage systems’ complexity, *e.g.*, the creation of prediction models (GOULAO et al., 2012).

Making this study possible requires the following tasks: (i) mine concerns and code complexity associated with code smells from the history of software projects; (ii) agglomerate artifacts’ code complexity around concerns; (iii) analyze combinations between concerns and agglomerations of complexity through the evolution of three types of software project; and (iv) present a variation of our investigation in which we substitute types for domains of software as a way to group systems.

Our findings show that agglomerations of code complexity follow a non-uniform pattern through the evolution of projects when types of software are adopted as transverse dimension. They also reveal that domains of software are partially better suited to spot more uniform agglomerations.

## C.1 STUDY DEFINITION

In this section, we present the definition of this study. We explain two key concepts behind it, transverse dimensions and agglomerations, and the research questions that guided us.

### C.1.1 Transverse Dimension

We believe that there is an optimal way to arrange software projects to favor the identification of concerns. Specifically, we conjecture that it is possible to enhance our investigations and findings after choosing a transverse dimension. We define “transverse dimension” as a classification schema which we use to mine concerns by joining systems’ historical data together. In the context of our research, we cover two transverse dimensions: Types and Domains of Software.

Table C.1 explains how we differentiate types and domains of software. We follow a strategy that distinguishes “application domain” from “programming domain” (O’BRIEN; BUCKLEY; SHAFT, 2004). “Application domain” (or “domains of software”, as we call it) refers to the context of the problem that is addressed by a piece of software, while “programming domain” (or “types of software”, as we refer to) concerns itself with technical details of implementing applications.

**Table C.1** Types and Domains of Software (CARVALHO; NOVAIS; MENDONÇA, 2020)

Dimension	Definition	Examples
Types	Systems which are similar to each other regarding targeted development platforms and architecture	Android Mobile Apps, Client-Server Applications
Domains	Systems that either share the same context of use or fulfil analogous sets of features	E-commerce Systems, Graph Databases, Chatbots

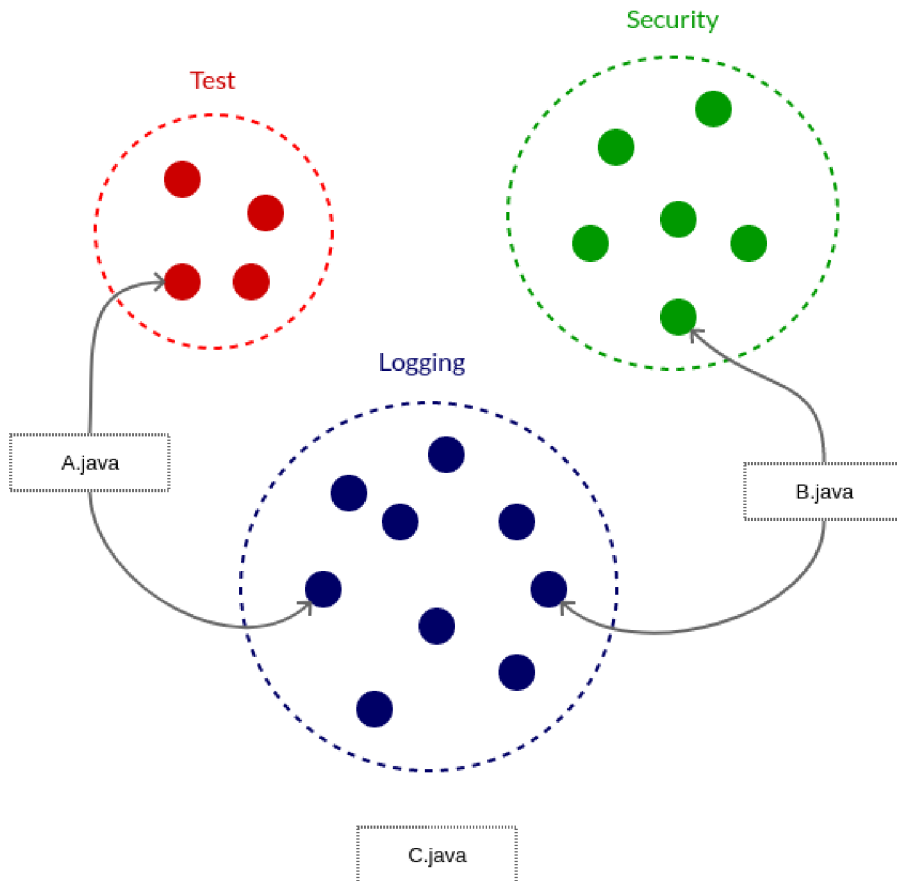
As we conducted our research, we carefully analyzed contexts in which the use of either types or domains of software was more/less adequate to mine concerns. This means: not only we take advantage of the metadata about components that developers embed in systems, but we also consider different strategies to group projects according to their types and domains while seeking to maximize the extraction and analysis of concerns.

### C.1.2 Agglomerations

We resorted to agglomerations to organize our dataset before performing analyses. Our definition of agglomerations stems from Oizumi *et al.*’s concept of “semantic agglomerations” (OIZUMI *et al.*, 2016): agglomerations are situations in which different source code artifacts affected by design problems address a particular concern. We are interested in the pieces of information systems (*e.g.*, \*.java files in java-oriented software projects) that contains design problems (*e.g.*, code complexity caused by the presence of code smells)

and, at the same time, implement a concern (*e.g.*, “Logging”). Next, we exemplify our use of agglomerations.

Figure C.1 illustrates three agglomerations. They are named after concerns found in a software project (project “P”). This means: by mining “P”, we find out that its developers automated “Test”, “Security”, and “Logging”. The figure shows the relationship between the agglomerations and three artifacts of “P”: “A.java”, “B.java”, and “C.java”. “A.java” takes part in the “Test” agglomeration (in red) because it automates tests and hosts a design problem. “A.java” logs some of its routines, so we also add it to the “Logging” agglomeration (in blue). As “B.java” implements “Logging” and contains a design problem, we add it to the “Logging” agglomeration as well. Developers inserted some routines related to “Security” in “B.java”, so we associate it with the “Security” agglomeration (in green). We are not interested in “C.java” as it either does not implement any of the three concerns or it does not contain a design problem. Consequently, “C.java” is not part of any agglomeration.



**Figure C.1** Agglomerations of Project “P” (CARVALHO; NOVAIS; MENDONÇA, 2020)

In (CARVALHO; NOVAIS; MENDONÇA, 2018), we examined the combinations between different types of software and agglomerations. We based our analysis on previous work (OIZUMI et al., 2014)(OIZUMI et al., 2015)(OIZUMI et al., 2016)(VIDAL et al.,

2016)(DÓSEA; SANT’ANNA; SILVA, 2018) which studied agglomerations of code smells. As a result, we found out that one possible way to visualize and analyze agglomerations is to stratify them as cases of similarities. **Similarities** comprise concerns which are shared by different types of software. **Similarities** can be divided into **full similarities** and **partial similarities**. Next, we explain and exemplify these types of agglomerations.

Figure C.2 exhibits the relationship between concerns and software projects grouped according to their types: distributed, service and mobile. **Full similarities** include cases in which common concerns agglomerate design problems for all of the mentioned types. “Test” is one example of this type of agglomeration. This means: we found out that the “Test” concern is associated with instances of code smells in the projects categorized as services. We also spotted the same association in the distributed and mobile systems. “Serialization”, “Database Connectivity”, “Network Access”, “Service-Orientation”, “Data Format Processing”, and “Stream Processing” are other examples of **full similarities**. **Partial similarities** agglomerate design problems for only a subset of the types. “Code Optimization” and “Mocking” are examples of this type of similarity because we found both concerns in distributed and mobile projects, but we did not find them in the services.

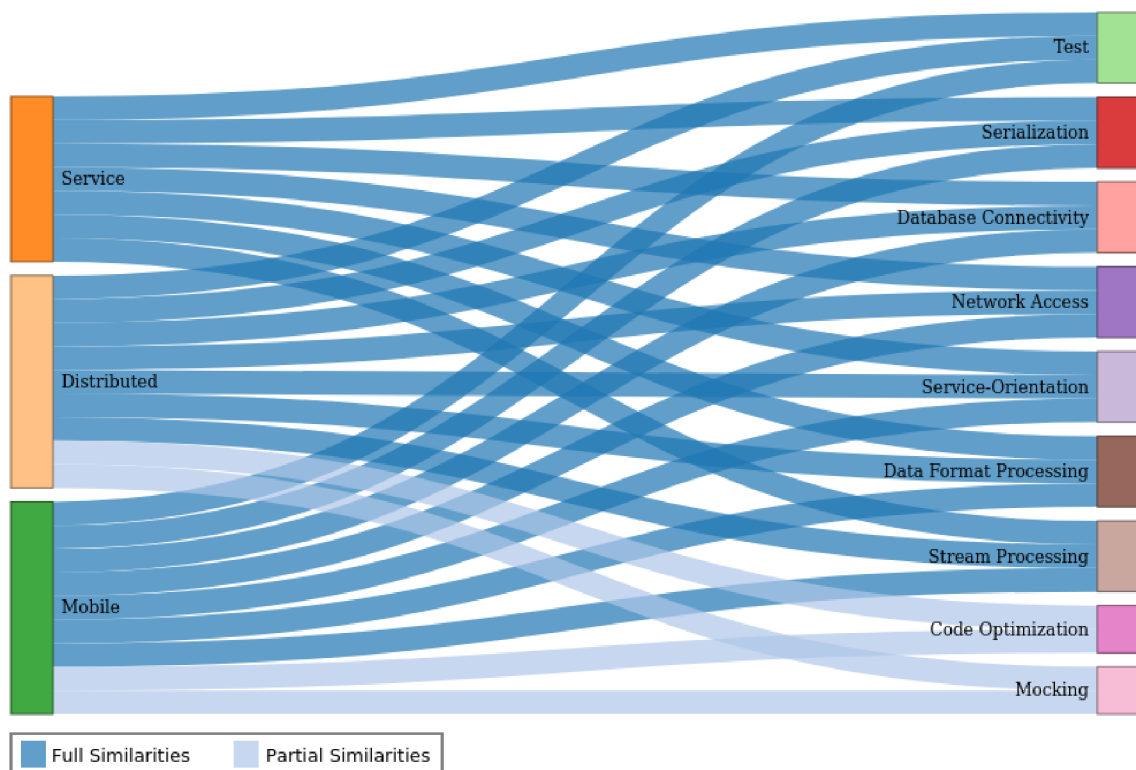
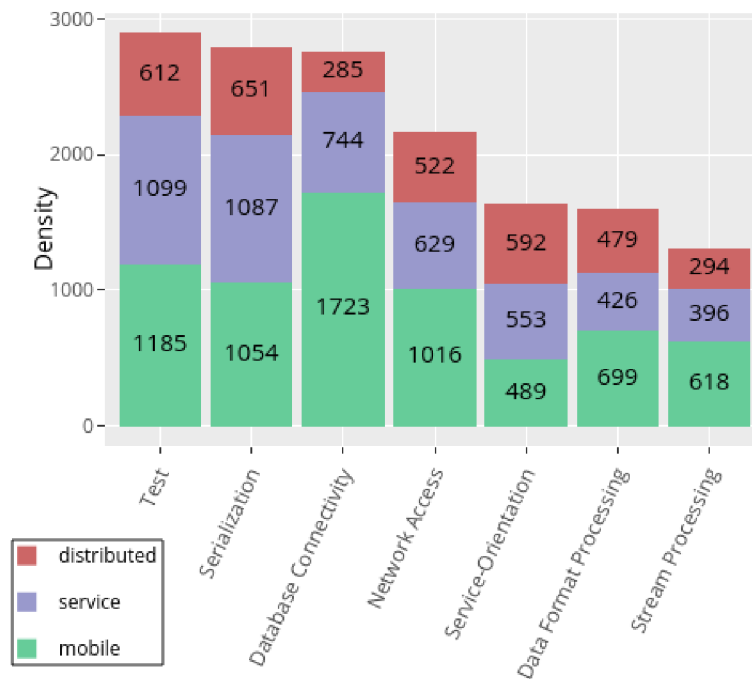


Figure C.2 Types of Similarities

We need to measure the size of the agglomerations as a way to compare them. We have resorted to the use of **density**. Density is an indirect metric that we calculate to compare the agglomerations and to produce visual clues (charts) about how strong/weak

is the association between design problems and agglomerations. Figure C.3 shows how we use the density of god classes (y-axis) to mine **full similarities** (x-axis) from systems<sup>2</sup>. In the figure, we used the Lines of Code (LOC) metric to measure the density of the agglomerations. Considering, for instance, the “Test” concern, we found cases of source code artifacts that implement “Test” in all three types of software, and some of these artifacts encapsulate god classes. Specifically, we agglomerated 1185 LOC from god classes of mobile applications, 1099 LOC from god classes of service-oriented projects, and 612 LOC from god classes found in distributed systems (the y-axis shows the total amount of LOC for the entire set of god classes). In other words, the developers of all the three types of software which we examined implemented tests in some source code artifacts, and the first vertical bar in the figure shows the density (or the strength) of the association between the test concern and the code smell that these artifacts host: god class. The same happens for other concerns presented in Figures C.3 and C.4. In Section we explain how we specialized the **density** metric to measure code complexity.



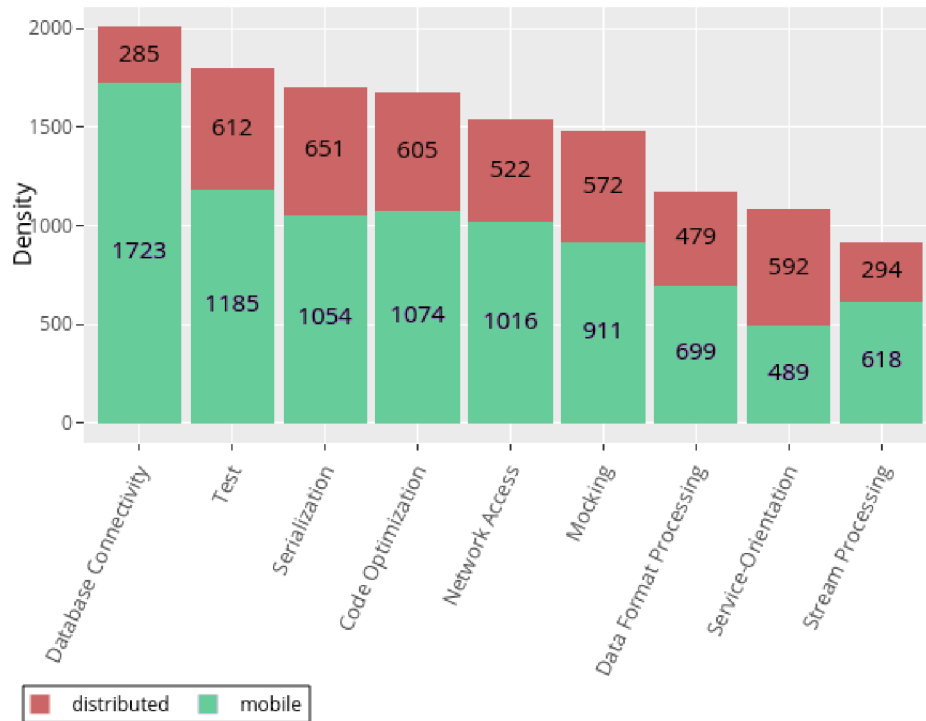
**Figure C.3** Similarities between Software Types and Concerns (Density by God Class) (CARVALHO; NOVAIS; MENDONÇA, 2018)

Figure C.4 shows **partial similarities** for distributed and mobile software projects. It has the same agglomerations presented in previous example (Figure C.3) and two new ones which are particular to these two types of software: “Code Optimization” and “Moking”. These two concerns are particular to this subset of software projects. This

<sup>2</sup>Table C.2 informs the name of the systems, the interval of time during their evolution and the number of source code artifacts that we mined from them



means: the service-oriented projects did not contain any artifact that implemented either “Code Optimization” or “Moking” and hosted god classes.



**Figure C.4** Partial Similarities between Distributed and Mobile Software Projects (Density by God Class) (CARVALHO; NOVAIS; MENDONÇA, 2018)

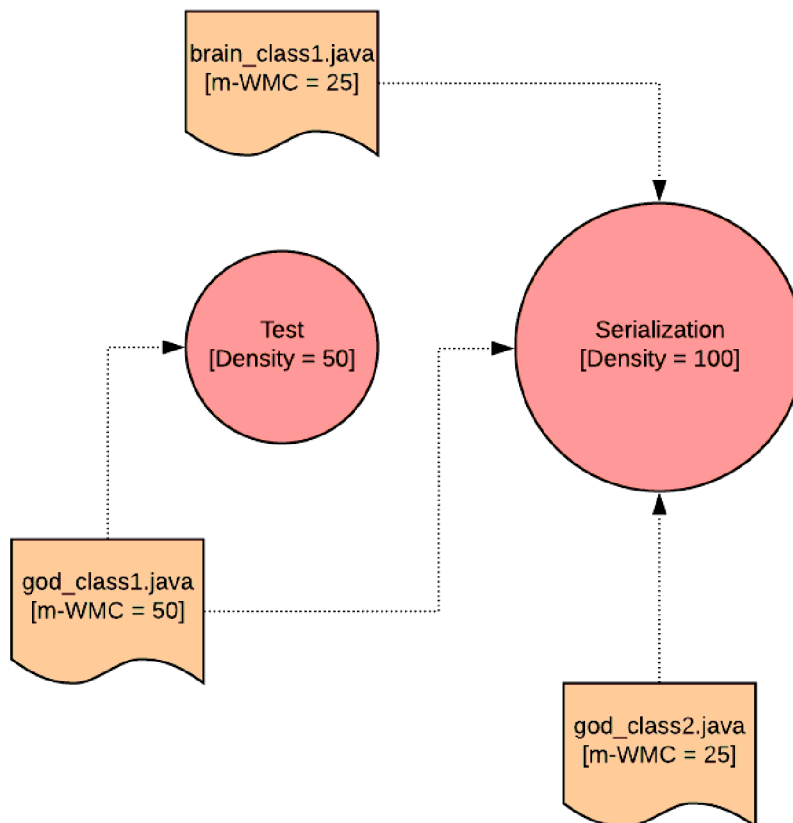
The charts exhibited in Figures C.3 and C.4 show a top-level view of how software projects agglomerate design problems around concerns. As a consequence, they do not show details about the evolution of the agglomerations. By not displaying the complete history of how similarities evolve through time, the graphics may lead observers into perceiving the agglomerations as uniform, or perfectly seasonal. We see this as a risk for our analysis: a top-level view of the data may hide discrepancies and cases of non-uniformity that only a detailed analysis of the evolution can reveal.

### C.1.3 Code Complexity Agglomerations

We have added routines to AKS to measure the degree to which each concern agglomerates design problems. As a result, AKS is capable of calculating the density of code complexity agglomerations from the Weighted Methods per Class (WMC) metric. In the context of this study, WMC’s value is obtained from the sum of the cyclomatic complexity of all methods within smelly classes (LANZA; MARINESCU, 2010). We took this decision after analyzing related researches which affirm that code complexity increases significantly in software artifacts that are affected by smells (LANZA; MARINESCU, 2010)(FONTANA et al., 2013)(VELIOGLU; SELCUK, 2017). As AKS depends on RM to mine software projects, we are restrained to the set of code smells that it can detect: god and brain

classes. Considering that RM’s god and brain classes detection rules naturally test the values of WMC against thresholds <sup>3</sup>, we guarantee that our study’s findings are based on relevant cases of code complexity.

The fictitious example in Figure C.5 illustrates how AKS calculates the density of code complexity agglomerations. AKS finds two instances of god classes in “god\_class1.java” and “god\_class2.java” and one instance of brain class in “brain\_class1.java”. These are artifacts of project “P”. “God\_class1.java” is associated with both “Test” and “Serialization” concerns. “God\_class2.java” and “brain\_class1.java” are associated with the “Serialization” concern only. For each concern, AKS obtains the normalized value of the density by calculating the mean-WMC (m-WMC) of classes. For example, AKS normalizes the density of “Test” by dividing the sum of WMC of all instances of “god\_class1.java” by the number of times the artifact appeared in the evolution of “P”. Then, AKS adds the m-WMC of “god\_class1.java” (50) to the density of “Test”. Similarly, AKS sums the m-WMC of “god\_class2.java” (25), “god\_class1.java” (50), and “brain\_class1.java” (25) to determine the density of “Serialization” (100).



**Figure C.5** Complexity-based Density of Agglomerations of Project “P”

<sup>3</sup>More information about RM’s code smells detection rules can be found in (<https://github.com/visminer/repositoryminer/wiki/Available-Code-Smells>)

### C.1.4 Research Questions

We looked forward to answer the following research questions:

**RQ1:** *Does source code complexity follow a uniform pattern as it agglomerates around concerns through the evolution of different types of software?*

RQ1 is the main question and seeks to understand if non-uniformity of agglomerations is either the norm or the exception during the evolution of software projects. We used AKS to mine 30921 source code snippets from the projects described in Table C.2 to answer this question.

**RQ2:** *Is there any other transverse dimension that can produce cases of more uniformly distributed agglomerations of complexity through the evolution of software projects?*

By answering **RQ1**, we examine the effects of a particular transverse dimension on the agglomerations of code complexity: types of software, but there is a possibility that domains of software is another fit candidate to agglomerate design problems.

We used the projects described in Table C.3 to answer **RQ2**. AKS analyzed 39485 artifacts through the historical data of the database systems. We also ran a statistical analysis on the dataset to find correlative associations between the agglomerations mined from the database projects. The statistical examination is another attempt to observe cases of uniformity through the evolution of the grouped systems. If the agglomerations of two (or more) databases correlate around the same concerns, this can be seen as an opportunity for defining common strategies to manage both projects' complexity. The opposite might indicate that assembling software projects under a transverse dimension is not fully advantageous. In other words, the use of transverse dimensions is more beneficial to developers if they are able to perceive a uniform pattern in the way how a design problem affects grouped projects through time. As another example, developers may try to understand problems of systems being currently developed after revisiting the past of grouped software projects. Again, this is only beneficial, if they can perceive a uniformity through the historical data.

Figure C.6 summarizes the activities of this study. In the first phase, we agglomerate instances of code complexity. As a result, we end up with agglomerations stratified as similarities and dissimilarities. We then divide the similarities into time series of systems' releases/versions to analyze how the agglomerations evolve. Answering **RQ1** has the purpose of knowing if splitting the dataset along a progression of releases/versions reveals that the agglomerations are uniformly distributed through time. If not, we want to test if the adoption of a different transverse dimension (domains of software) tend to produce more uniform agglomerations (this being the purpose of **RQ2**).

We highlight that the software projects that we group as either “types” or “domains”, and which we used in our analyses are not mutually exclusive in terms of the concerns they host. This means: it was not part of our selection criteria to fully dissociate projects regarding the concerns that developers embedded in them. Consequently, a distributed system may include some concerns which service-oriented systems also have.

We evaluated the Titan project as both a type (distributed) and a domain (graph database) of software. So, isolating a system within a particular transverse dimension was not part of our selection strategy either.

**Table C.2** Analyzed Projects (Transverse Dimension: Types of Software) (CARYALHO; NOVAIS; MENDONÇA, 2018)

Type	Project	Description	Period	Artifacts
Distributed	Genie <sup>a</sup>	Federated job orchestration engine developed by Netflix	2017-04 - 2018-01	2012
	Pinot <sup>b</sup>	A realtime distributed OLAP datastore	2016-01 - 2017-12	7586
	ShardingSphere <sup>c</sup>	An open-sourced distributed database middleware solution suite	2016-05 - 2018-02	2693
Service	Titan <sup>d</sup>	A database optimized for storing and querying large graphs	2012-06 - 2015-09	2698
	Zipkin <sup>e</sup>	A distributed tracing system	2017-06 - 2018-01	1577
	Cellbase <sup>f</sup>	NoSQL DB and Web Services to access biological data	2016-09 - 2017-11	1193
Mobile	GeoApi <sup>g</sup>	New York Senate Geopolitical Service API	2013-03 - 2018-03	1076
	OHDSIWeb <sup>h</sup>	Services for the Observational Health Health Data Sciences and Informatics	2016-09 - 2017-10	1450
	OpenLegislation <sup>i</sup>	New York Senate Legislation Service API	2013-02 - 2018-04	2812
	OpenMRS <sup>j</sup>	OpenMRS REST Web Services Module	2017-02 - 2017-10	2068
Mobile	IrcCloud <sup>k</sup>	A Chat on IRC for Android	2013-09 - 2018-05	528
	OkHttp <sup>l</sup>	Android client for the OkHttp network optimization suite	2013-06 - 2018-02	965
	NextCloud <sup>m</sup>	Android version of the Next Cloud Application	2017-10 - 2018-01	1331
	Retrofit <sup>n</sup>	Type-safe HTTP client for Android and Java	2013-09 - 2016-01	392
	Signal <sup>o</sup>	A messaging app for simple private communication with friends	2014-12 - 2018-04	2540

<sup>a</sup> <https://github.com/Netflix/genie>, <sup>b</sup> <https://github.com/linkedin/pinot>, <sup>c</sup> <https://github.com/sharding-sphere/sharding-sphere><sup>d</sup> <https://github.com/thinkarewinius/titan>, <sup>e</sup> <https://github.com/openzipkin/zipkin>, <sup>f</sup> <https://github.com/opench/cellbase><sup>g</sup> <https://github.com/nysenate/GeoApi>, <sup>h</sup> <https://github.com/OHDSI>, <sup>i</sup> <https://github.com/nysenate/OpenLegislation><sup>j</sup> <https://github.com/openmrs/openmrs-module-web-services-rest>, <sup>k</sup> <https://github.com/irccloud/android><sup>l</sup> <https://github.com/square/okhttp>, <sup>m</sup> <https://github.com/nextcloud>, <sup>n</sup> <https://github.com/square/retrofit><sup>o</sup> <https://github.com/signalapp/Signal-Android>

**Table C.3** Analyzed Projects (Transverse Dimension: Domains of Software) (CARVALHO; NOVAIS; MENDONÇA, 2020)

Domain	Project	Description	Period	Artifacts
Graph	JanusGraph <sup>a</sup>	Highly scalable graph database	2017-04 - 2018-10	5657
	Neo4J <sup>b</sup>	High performance graph store with all the features expected from a robust database	2018-09 - 2018-12	26497
	Titan <sup>c</sup>	Database optimized for storing and querying large graphs	2012-06 - 2015-09	3570
Time Series	OpenTSDB <sup>d</sup>	Distributed, scalable TS database	2015-11 - 2018-12	1440
	KairosDb <sup>e</sup>	Fast distributed scalable TS database written on top of Cassandra	2015-11 - 2018-11	1884
	Timely <sup>f</sup>	Time series database that provides secure access to time series data	2016-06 - 2017-08	827

<sup>a</sup> <https://github.com/JanusGraph/janusgraph><sup>b</sup> <https://github.com/neo4j/neo4j><sup>c</sup> <https://github.com/thinkaurelius/titan><sup>d</sup> <https://github.com/OpenTSDB/opentsdb><sup>e</sup> <https://github.com/kairosdb/kairosdb><sup>f</sup> <https://github.com/NationalSecurityAgency/timely>

Figure C.2 shows three types of software: service, distributed, and mobile. We spotted the presence of “Service-Orientation” (the concern) in all of them. So, it is arguable that we could have categorized the distributed and mobile systems as services as well because they externalize some of their functionalities as services. We regard this type of concern as an attempt to add a composition of concerns to software projects. Developers often benefit from libraries that provide many features at once, *i.e.*, function compositions that fulfill several concerns (BUSCH et al., 2019). Including “Service-Orientation” (*i.e.*, a subsystem of services) in distributed and mobile systems is an example of this. In our analysis, we have not differentiated more complex concerns (*e.g.*, “Service-Orientation”, “Test”) from others that provide simpler functionalities (*e.g.*, “Logging”). We decided to stick to the way how their developers explicitly classified them according to descriptions that we found in the projects’ github repositories (footnotes in Tables C.2 and C.3).

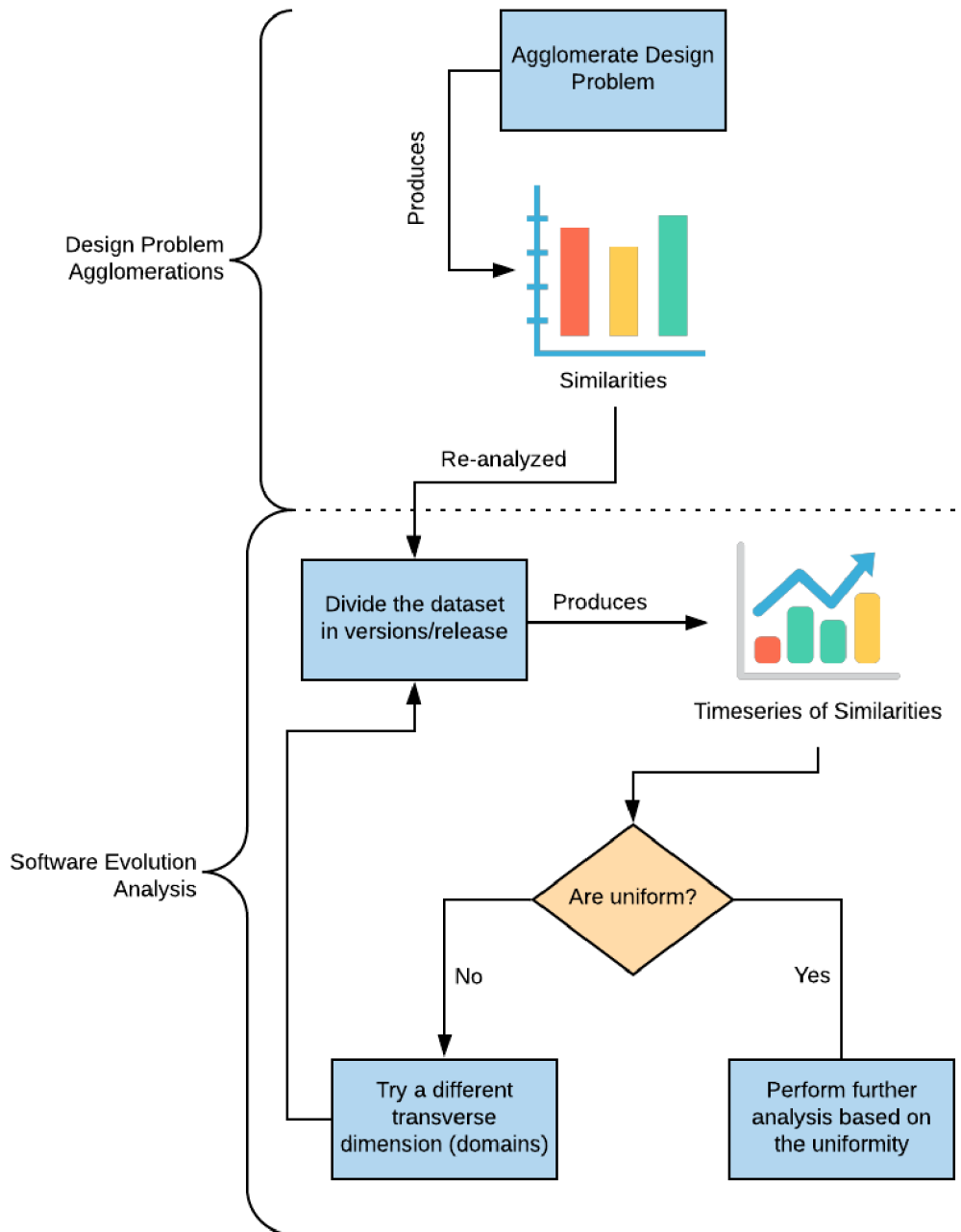


Figure C.6 Summary of this Study

## C.2 RESULTS

All visualizations shown in this section (from Figure C.7 to Figure C.11) follow the same pattern. They show a composition of charts and each chart displays evolutionary data extracted from one software project of one specific type or domain of software. The x-axis is divided in versions which AKS mined from the software projects. Every single chart shows a number of five versions per project. The y-axis quantifies the density of

the agglomerations shown in the bar plots. In this case, we used a complexity metric (m-WMC) to calculate the density (as explained in Section C.1.3).

Figure C.7 exhibits the evolution of code complexity agglomerations considering the concerns which are common to three types of software: distributed, service-oriented, and mobile systems. The chart displays sources of non-uniformity: some projects do not contribute for increasing the density of the agglomerations. For instance, genie, signal, and openmrs do not amount density for any of the concerns: “Data Format Processing”, “Database Connectivity”, “Network Access”, “Service-Orientation”, “Serialization”, and “Test” (the complete list of concerns can be found in Appendix D). Other projects produce agglomerations whose densities are not constant through evolution. Shardingsphere, titan, ircloud, and geoapi are examples of inconstant projects. We consider that the distributions of agglomerations through the evolution of these project are non-uniform. On the other hand, the evolution of pinot, zipkin, nextcloud, okhttp, retrofit, cellbase, ohdsiweb, and openlegislation agglomerate densities throughout their evolution. We call these cases uniform agglomerations.

Next, we present the results of the analysis we performed on partial similarities. Figure C.8 shows the evolution of agglomerations mined from distributed and service-oriented software projects. “Dataset Processing”, “Logging”, “Messaging”, “Process Execution”, and “Programming Utilities” are concerns which are common to these two types of software. The non-uniformity resides in the fact that genie and openmrs do not show any density for these concerns through their evolution. Additionally, the density is not detected considering some specific versions of other projects. This is the situation of shardingsphere, titan, and geoapi. However, projects like pinot, zipkin, cellbase, ohdsiweb, and openlegislation show a more constant flow of agglomerations as they evolve.

Figure C.9 exhibits the partial similarities between mobile and service-oriented software projects. The list of concerns includes the ones exhibited in Figure C.7 and two others that are particular to these two types of software: “Security” and “Stream Processing”. Openmrs shows no agglomeration through its evolution. Agglomerations obtained from ircloud, signal, geoapi, and openlegislation are non-uniform. Nextcloud, ohdsiweb, okhttp, retrofit, and cellbase show more uniform distributions.

Figure C.10 shows partial similarities obtained from the comparison between distributed and mobile software projects. The projects share only one concern: “Mocking”. Genie and Signal do no agglomerate any density through their evolution. Shardingsphere, titan, and ircloud do not show a uniform distribution of agglomerations as they evolve. Pinot, zipkin, nextcloud, and retrofit are more uniform in terms of how the densities are distributed through the historical data.

Now, we present the results of our analysis of the non-relational database systems. Figure C.11 shows the collection of full similarities mined from the projects described in Table C.3. One noticeable aspect of the figure is that the projects share more common concerns if compared to our previous analysis regarding types of software (as seen in Figure C.7). A visual inspection of the figure indicates that the agglomerations are more uniformly distributed through the evolution of the software projects.

We further examined the data statistically to confirm the uniformity. Considering our main goal of helping developers to manage design problems, we consider that it is desirable

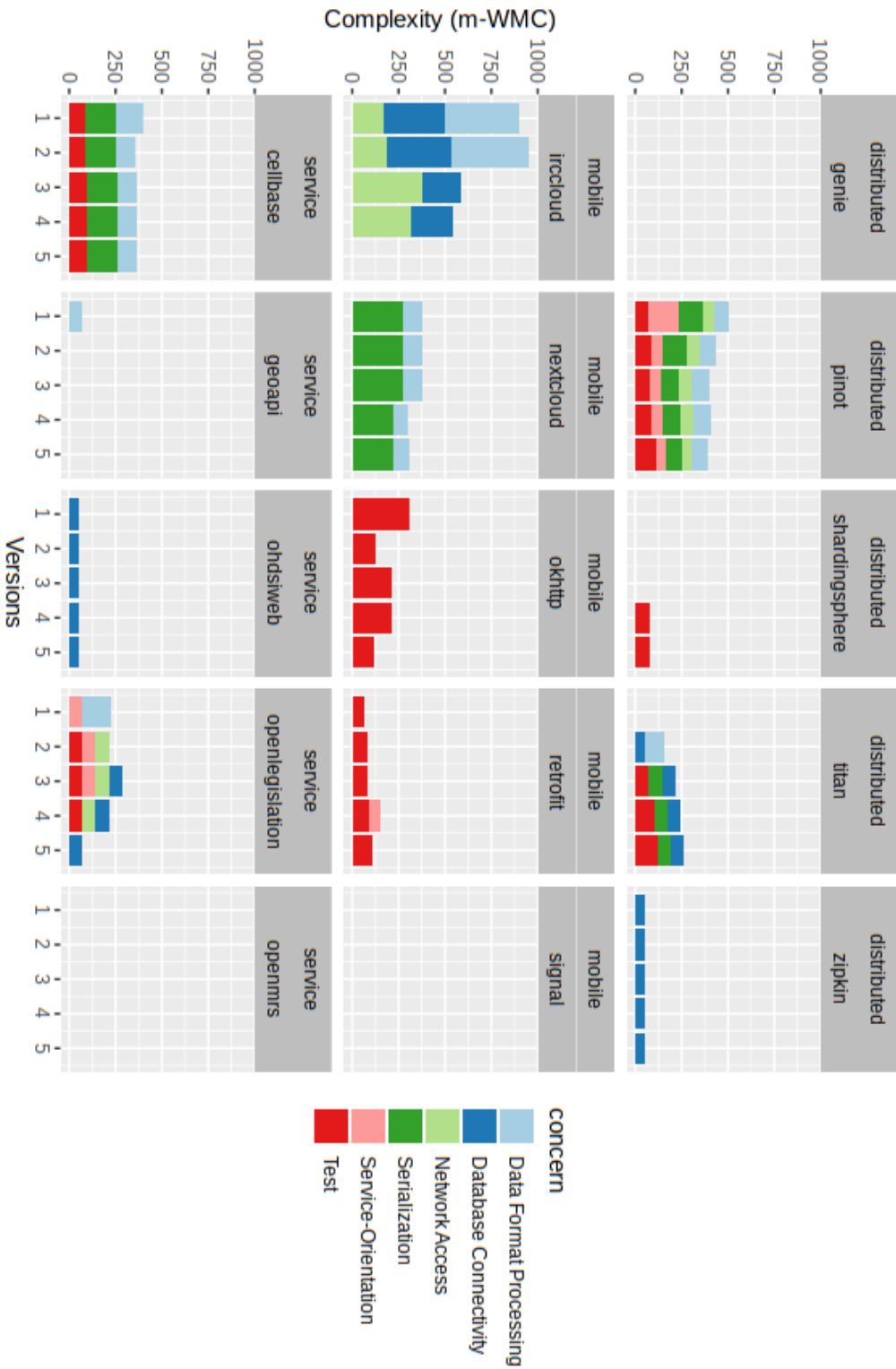
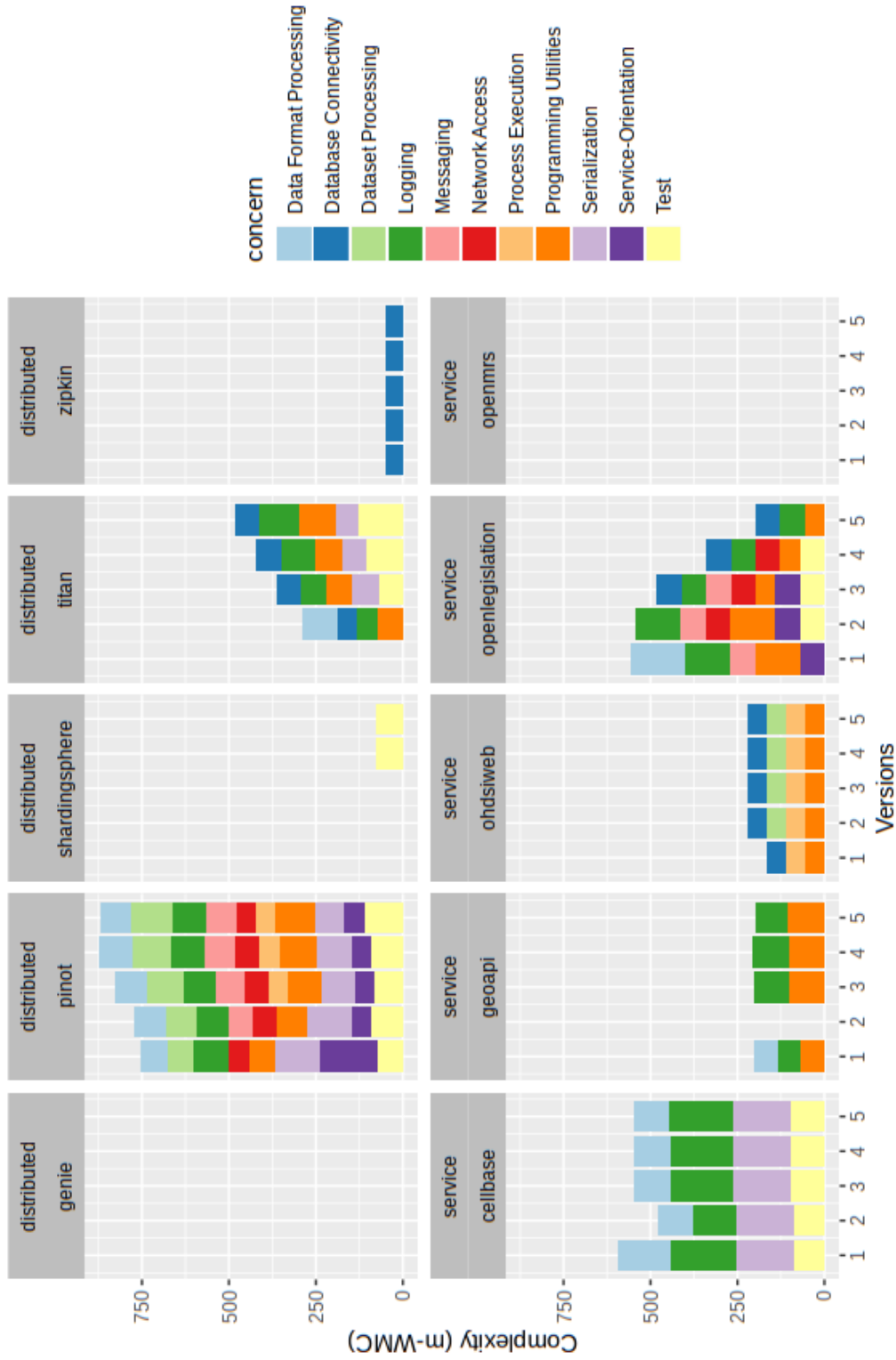
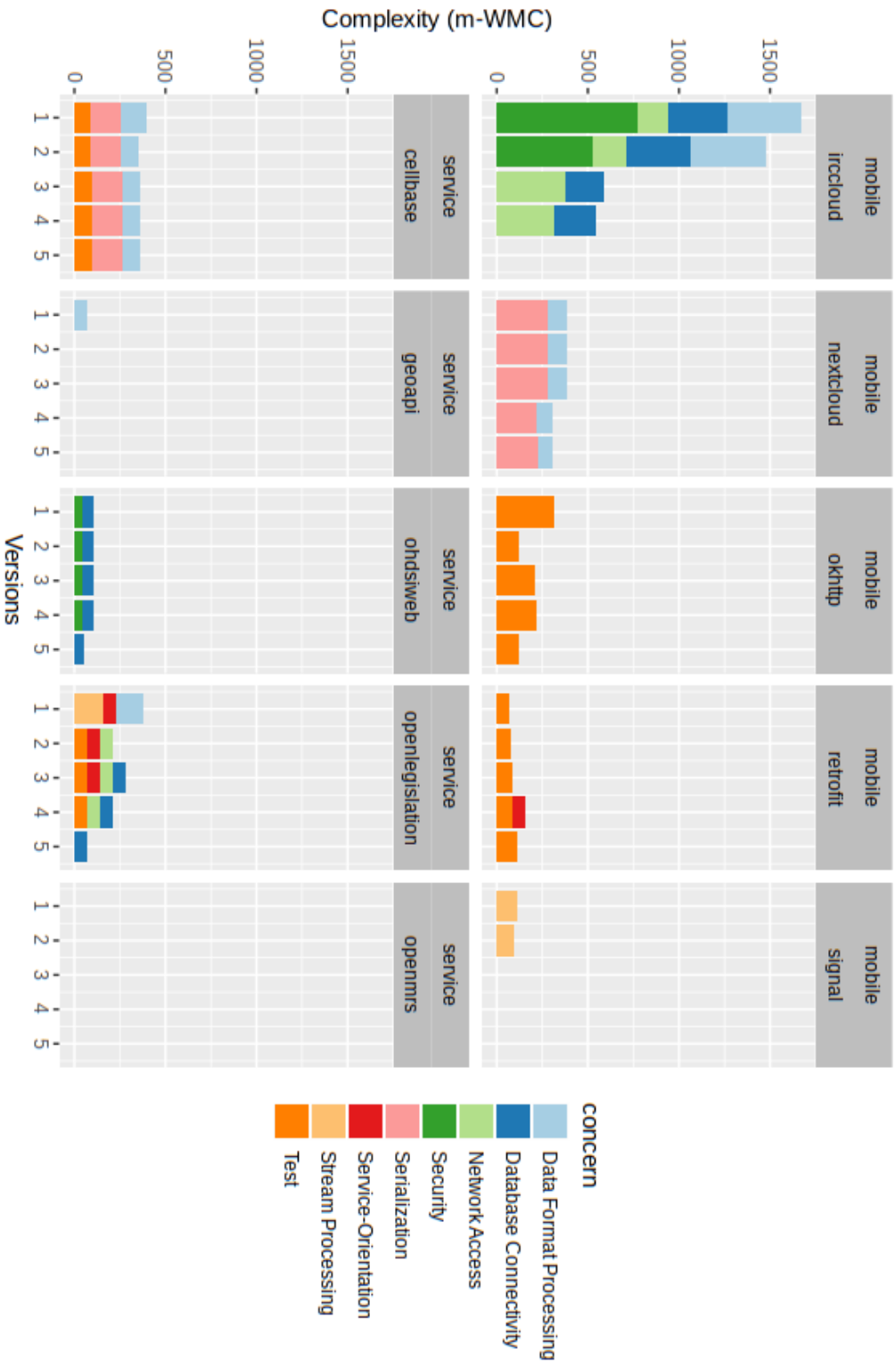


Figure C.7 Uniformity of Full Similarities – Transverse Dimension: Types of Software (CARVALHO; NOVAIS; MENDONÇA, 2020)





**Figure C.8** Uniformity of Partial Similarities: Distributed X Service-Oriented – Transverse Dimension: Types of Software (CARVALHO; NOVAIS; MENDONÇA, 2020)



**Figure C.9** Uniformity of Partial Similarities: Mobile X Service-Oriented – Transverse Dimension: Types of Software (CARVALHO; NOVAIS; MENDONÇA, 2020)

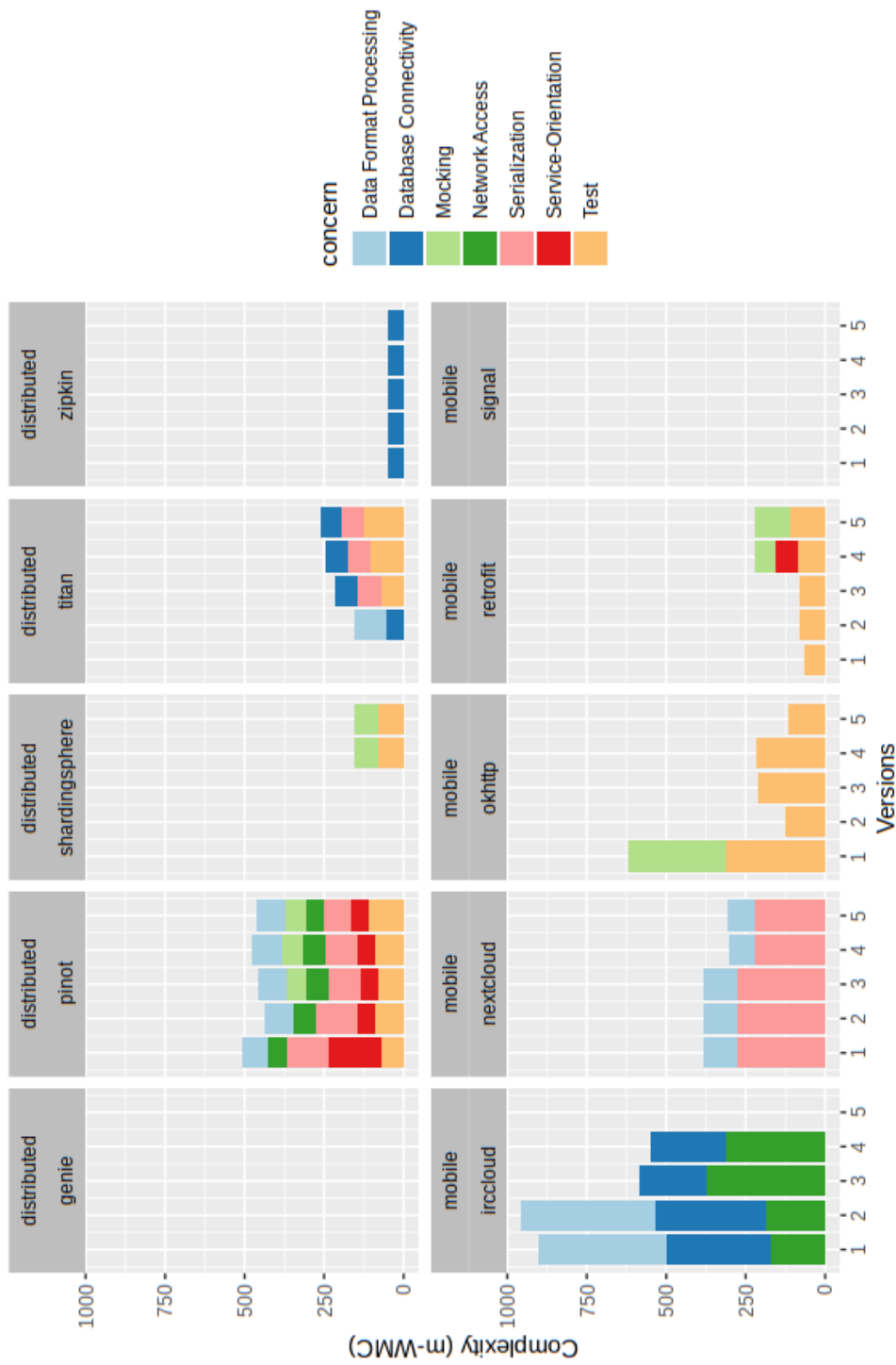


Figure C.10 Uniformity of Partial Similarities: Distributed X Mobile – Transverse Dimension: Types of Software (CARVALHO; NOVAIS; MENDONÇA, 2020)

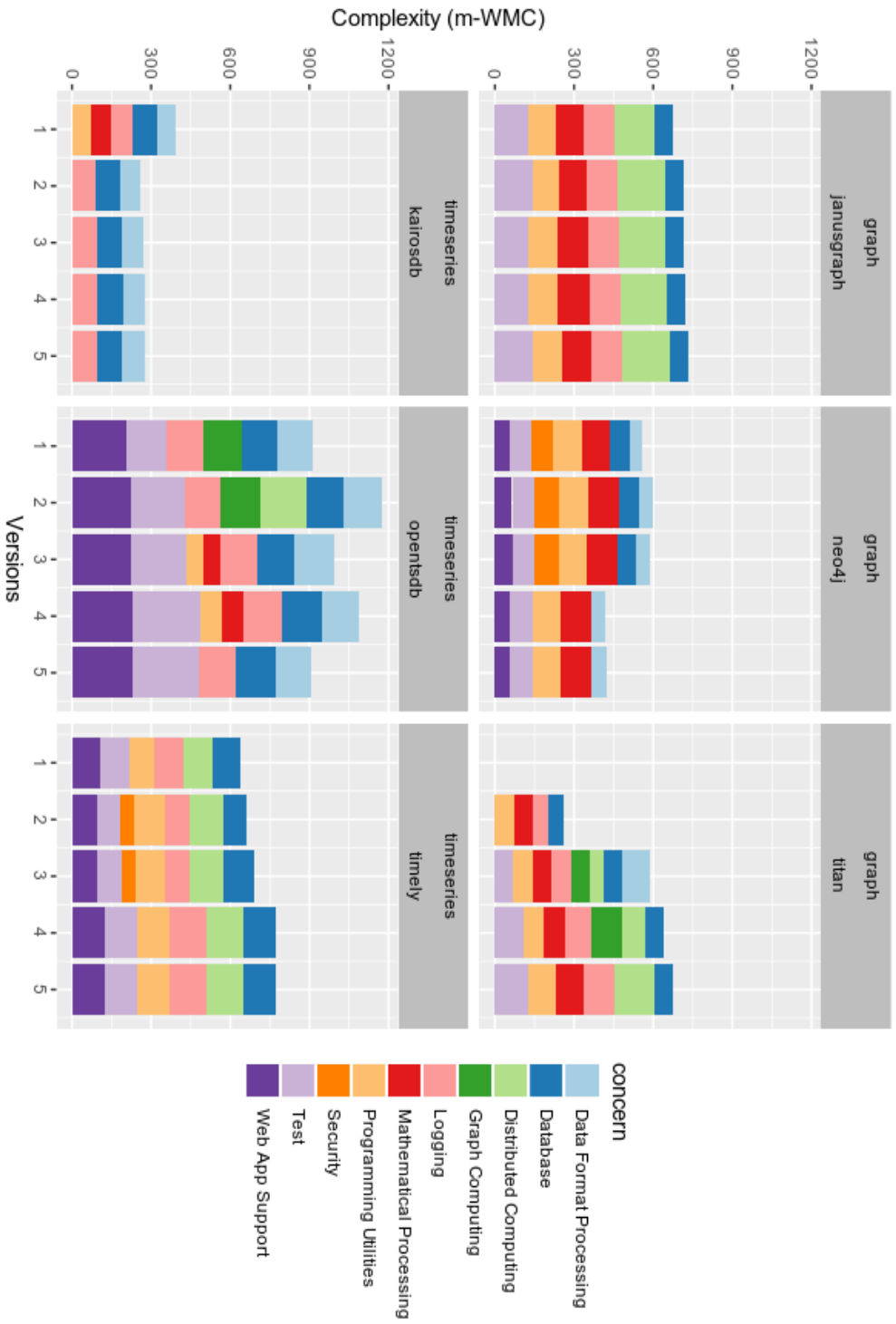


Figure C.11 Uniformity of Full Similarities – Transverse Dimension: Domains of Software (CARVALHO; NOVAIS; MENDONÇA, 2020)

that the agglomerations are uniformly distributed through evolution. For instance, “Test” is uniformly distributed, as it affects many versions of the projects according to a visual analysis of Figure C.11, but it is important to know if the “Test” agglomerations can correlate with each other through a time series of versions/releases.

Tables C.4 and C.5 contain a series of correlational tests pertaining all associations between the agglomerations shown in Figure C.11. For each concern shared by two distinct projects we tested if the evolution of the code complexity agglomerations occurred in concomitance. The data is not normally distributed, so we relied on a non-parametric analysis method (spearman).

The tests exhibited in Table C.4 and in its continuation (Table C.5) revealed a multitude of results in which high correlations are mixed with low ones. For example, “Test” produced significant correlations between certain pairs of software projects (*e.g.*, janusgraph–titan, janusgraph–opentsdb), but no significant correlation between others (*e.g.*, janusgraph–neo4j, titan–timely). As the latter are not exceptional cases, we cannot assume that comparing the evolution of code complexity from any two paired software projects reveals a uniform pattern.

### C.3 THREATS TO VALIDITY

We now discuss the threats to validity which we identified in this study:

**Construct validity:** as illustrated in Figure 3.4, we mined concerns from third-party components injected in software projects. Later on, our approach used the “import” directive to associate source code artifacts with concerns. However, we cannot guarantee the imported components are extensively used by the artifacts they are injected in. We have not empowered AKS with features to reject cases in which the imported components are scarcely used to implement a concern. Such features would also have to favor cases in which the bound between components and artifacts spreads over many lines of code or causes a more significant impact. Therefore, using components injection to implement the concept of agglomerations is a potential source of inconsistencies regarding precision in our data. One way to circumvent this limitation requires embedding the mentioned routines in AKS. This will help us to precisely spot situations in which the influence of components (and the association between concerns and agglomerations) is too diluted to be taken in account. Consequently, this might refine our dataset and observations.

**Internal validity:** even though developers may use a given concern extensively through an artifact, it may be the case that the concern is not alone. Developers may feel like importing several components to support the implementation of many concerns. This imposes a problem: uncertainty regarding the degree to which we can relate the occurrence of code complexity (or any other design problem) to a concern. For instance, source code artifacts are not likely to focus on the use/implementation of the “Logging” concern in isolation. “Logging” tends to play an auxiliary role as developers often implement it as a way to register the activities of other concerns (*e.g.*, logging of steps while accessing a database and the subsequent processing of resultsets). On the other hand, they may specialize other artifacts in performing “Test” routines. This comes from the fact that developers frequently create tests to deal with specific sets of a system’s functionalities.

**Table C.4** Concern-Based Correlations between Projects (CARVALHO; NOVAIS; MENDONÇA, 2020)

<b>Project</b>	<b>Project</b>	<b>Concern</b>	<b>Correlation</b>
janusgraph	neo4j	Database	0.44
	neo4j	Mathematical Processing	0.23
	neo4j	Test	0.28
	titan	Database	0.45
	titan	Logging	0.04
	titan	Mathematical Processing	0.51
	titan	Distributed Computing	0.37
	titan	Test	0.64
	kairosdb	Database	0.04
	kairosdb	Logging	0.22
	kairosdb	Mathematical Processing	0.55
	opentsdb	Database	0.31
	opentsdb	Logging	0.45
	opentsdb	Test	0.67
	timely	Database	0.45
	timely	Logging	0.56
	timely	Distributed Computing	0.09
timely	Test	0.70	
neo4j	titan	Database	0.01
	titan	Mathematical Processing	0.18
	titan	Test	0.80
	kairosdb	Data Format Processing	0.01
	kairosdb	Database	0.24
	kairosdb	Mathematical Processing	0.17
	opentsdb	Data Format Processing	0.60
	opentsdb	Database	0.01
	opentsdb	Test	0.36
	opentsdb	Web App Support	0.71
	timely	Database	0.01
	timely	Test	0.80
timely	Web App Support	0.05	
kairosdb	opentsdb	Data Format Processing	0.68
	opentsdb	Database	0.22
	opentsdb	Logging	0.08
	timely	Database	0.25
	timely	Logging	0.25
opentsdb	timely	Database	0.05
	timely	Logging	0.11
	timely	Test	0.25
	timely	Web App Support	0.24

Continue in the next page...

**Table C.5** Concern-Based Correlations between Projects (Continuation) (CARVALHO; NOVAIS; MENDONÇA, 2020)

Project	Project	Concern	Correlation
titan	kairosdb	Database	0.25
	kairosdb	Logging	0.08
	kairosdb	Mathematical Processing	0.17
	opentsdb	Database	0.00
	opentsdb	Logging	0.13
	opentsdb	Test	0.09
	timely	Database	0.04
	timely	Logging	0.25
	timely	Distributed Computing	0.04
	timely	Test	0.11

The specialization stems from a good practice related to the automation of tests: single responsibility. This principle is better achieved when tests focus on either very few or on one single behavior of a system (RUNESON, 2006)(BOWES et al., 2017). Therefore, it is more likely that a given design problem can be associated with an artifact that implements “Test” (to test one feature of systems) than with one that hosts “Logging” (in combination with other concerns). We must comprehend the use that developers make of different concerns (in isolation vs in combination) to refine our findings.

We have investigated the impact of types of software as transverse dimension (CARVALHO; NOVAIS; MENDONÇA, 2018). We are now studying the effects of a new possible categorization of agglomerations: domains of software. We claim that our findings can be trusted, but we still have work to do before we are able to fully testify that a variation from types of software to domains of software grants uniformity. We can achieve this by finding and analyzing other related projects (*e.g.*, other graph and time series databases) or adding new sub-domains to our investigations (*e.g.*, relational databases, mathematical applications, health care systems).

It is also important to find ways to enhance the results of correlational tests as the one that we performed on pairs of database projects (Table C.4). This may require re-analyzing our dataset via statistical methods that favor non-seasonal/non-uniform data. Additionally, it is imperative to define strategies to pair the versions of different projects adequately. The versions of the databases that we tried to correlate may represent different development stages or levels of maturity. We must find a way to match projects’ versions to avoid comparing a system’s immature versions with another system’s more mature ones.

**External validity:** surely, the number of types and domains of software abounds beyond the ones we examined here. While we believe that our conclusions are assertive, we cannot say that they can embrace other projects or a different set of transverse dimensions. A generalization would require expanding our studies to consider other scenarios. With this purpose in mind, we have been improving our approach, automatizing AKS, and expanding our dataset. This can contribute for advancing and generalizing our stud-

ies toward other associations between software projects, design problems and transverse dimensions.

**Conclusion validity:** we have conjectured about the applicability of agglomerations to define strategies to manage the incidence of design problems in software projects. We have also cogitated that such applicability is more advantageous if the agglomerations are uniformly distributed through the evolution of projects. However, we must evaluate these conclusions in real (or near to real) situations. This must include gaining more knowledge about: (i) the actual degree to which developers would value an approach to analyze the association between design problems and concerns through the evolution of software projects; (ii) the impact of uniformity and non-uniformity in their analysis as they manage the incidence of design problems.

## C.4 DISCUSSION

With this study, we address the association between the code complexity caused by code smells and the evolution of concerns. We applied types and domains of software as transverse dimensions attempting to favor the uniformity of our dataset. We see uniformity as an important requirement to enable the use our method to develop strategies to mitigate design problems. Considering the results that we presented in Section C.2, we provide the following answers for our research questions:

**RQ1:** *Does source code complexity follow a uniform pattern as it agglomerates around concerns through the evolution of different types of software?*

By observing the distribution of agglomerations through the evolution of different types of software, non-uniformity is the pattern that full and partial similarities tend to follow. For instance, projects like Genie, OpenMRS, and Signal do not contribute for increasing the density of agglomerated instances of code complexity and the distribution of agglomerations obtained from projects like ShardingSphere and GeoApi is non-uniform.

Alternatively, we can also say that revealing projects like Genie, OpenMRS, and Signal do not concentrate densities for agglomerations of design problems may actually be a positive finding. This means: AKS found no instance of god and brain classes in these projects to associate code complexity with. This can be an indicator of the good quality of their source code. The same can be said about ShardingSphere and GeoApi, as they did not contribute for the density of agglomerations. An approach to manage design problems may skip the analysis of these projects or relegate them to a less important rank of problematic systems. Meanwhile, developers can focus on the most significant cases, *e.g.*, Pinot and Celbase.

**RQ2:** *Is there any other transverse dimension that can produce cases of more uniformly distributed agglomerations through the evolution of software projects?*

Domains of software like the ones we analyzed (graph and time series databases) seem



prone to share more concerns. Most probably, this comes from the fact that a common context of use can lead developers to implement similar functionalities. The similarity can even cause the injection of the same (or nearly the same) third-party components.

Domains are also more inclined to produce cases in which agglomerations are uniformly distributed through the evolution of software projects. While this affirmation is assertive considering the visual inspection of projects' evolution (shown in Figure C.7), it fails under statistical analysis (Table C.4), *i.e.*, adopting domains of software as transverse dimension can show which shared concerns agglomerate instances of design problems when different software projects are in play, but it may not provide a statistical base for a fully advantageous application of our approach.

We believe that using domains of software as transverse dimension can grant a partial but important advantage in comparison to types of software. Domains are advantageous as they reveal more cases of concerns being shared by software projects. Additionally, the resulting agglomerations of design problems have a tendency to distribute themselves more uniformly through systems' evolution. Although failing to correlate under a statistical analysis can be seen as unfavorable, we do not regard this as so impacting that it may prevent developers from using our approach. This finding influenced us to group software projects according to their domains to conduct the studies described in Chapters 4 and 5.

Among future studies, we have the intention to circumvent the problems and limitations elicited in Section C.3. As well, we intend to perform other investigations. For instance, we would like to broaden the variability of transverse dimensions. We wonder if other transverse dimensions have potential to reveal more variations related to the way how design problems agglomerate around concerns. For instance, development methods might be a good addition to our studies. How are agglomerations distributed through the evolution of software projects which have adopted Test-Driven Development (TDD)(BAJAJ; PATEL; PATEL, 2015) as method? Compared to standardized classic methods, we expect to see TDD-based projects agglomerating more density for test-related concerns right from the beginning of software evolution. Perhaps, projects that follow classical methods (*e.g.*, cascade) (SOMMERVILLE, 2010) may have a tendency to cluster design problems around this concern only during later development phases. If this is the case, design problems which frequently plague test artifacts will tend to appear at different moments depending on the chosen method. Developers can use this information to manage test-related design problems when the time is right.

In Section C.1.4 (last paragraph) we highlighted the possibility that embedding third-party components in software projects can lead to composition of concerns (BUSCH et al., 2019). We even mentioned the case of "Service-Orientation" being found in distributed and mobile systems as a way to implement subsystems of services. It seems possible to contrast the effects of using third-party components to inject simpler libraries (*e.g.*, to implement "Logging") from the embedding of more complex ones (*e.g.*, "Geospatial Processing", "Bioinformatics"). We can categorize the later as frameworks and associate them with attempts to add collections of features and concerns that are common to specific domains of software (VALE et al., 2016). As another future work, it is important to evaluate if our approach is adequate to spot developers' strategies to use frameworks

to compose interrelated concerns.

We must also focus on making our dataset more reactive to developers’ actions. We are not sure that developers usually look around through the evolution of systems to find and manage code smells and the complexity caused by them. Thus, we are less likely to see a steady increase/decrease in the number of smells through time. Consequently, a pattern in the way how the complexity of smelly artifacts evolve may not be observed either. By not reacting to smells, developers may have other motivations to change the source code and, perhaps, the motivations reside in the concerns. Developers of graph databases may tend to focus on improving concerns that are important to this type of software: “graph computing”, “mathematical processing”, “distributed computing” (here, we are assuming these ones as examples of most important concerns for the sake of argument). As well, they may consider other concerns as less significant or some concerns will not demand many attention through time. We can raise a new hypothesis then:

*To systems grouped under a given transverse dimension there is a subset of concerns which most development effort centrepoint to. Tracking the evolution of such concerns can reveal the actual problems that developers deal with in a daily basis.*

Investigating the hypothesis may provide a more refined approach to help developers: instead of (firstly) selecting design problems and agglomerating problematic artifacts around concerns, we should better try to find artifacts that implement concerns which attract developers’ attention more often (*e.g.*, the number of commits affecting artifacts that implement “graph computing”, “mathematical processing”, and “distributed computing” may indicate this) and then identify the design problems that habitually affect these artifacts. This can be optimal as it might provide a reactive dataset about concerns that developers care the most while discarding/deprioritizing others which are of less impact.

## C.5 RELATED WORK

We highlight the following related work that has investigated agglomerations of design problems:

In a series of papers, Oizumi *et al.* (OIZUMI et al., 2014)(OIZUMI et al., 2015)(OIZUMI et al., 2016) investigated how to use code anomalies as the main design problem to address architectural and design problems in software projects. In their analyses, they have even chosen some software projects based on their types (*e.g.*, web frameworks and middlewares) and design styles (*e.g.*, N-layers and MVC);

Kazman *et al.* (2015) and Mo *et al.* (2015) interrelated software artifacts to address architectural design flaws through the evolution of analyzed projects. They point out that clustering source code files into special structures called Design Rule Hierarchy (DRS) can help developers to spot parts of systems that are more error and/or change-prone. However, they did not show any results regarding the analysis of concerns and their relationship with transverse dimensions;

Tufano *et al.* (2015) investigated *when* and *why* instances of code smells are introduced

in software projects. Their research resembles our own as both seek to understand how the appearance of code smells fluctuates through the evolution of software projects. Tufano *et al.* (2015) provided a broad study about the phenomenon with no focus on grouping the analyzed data around concerns or any specific transverse dimension;

Vidal *et al.* (2016) explored the use of agglomerations to prioritize the management of damaging smells. They also emphasize the applicability of architecturally-relevant concerns and the use of different design styles (*e.g.*, N-layers and MVC) as transverse dimensions to evaluate the effects of design problems on the quality of software projects. Both this and the previous work contributed to shape many of the concepts we have applied in our studies, but they did not analyze their results considering the impact of transverse dimensions as deeply as we have;

Dósea, Sant'anna and Silva (2018) evaluated the impact of design decisions on the definition of metrics for the analysis of source code. They also relied on concerns (or design roles, as they call it) from different types of software (*e.g.*, eclipse's plugins, android applications, web-based systems) to base their studies;

Our study either differs-from or expands the aforementioned ones by bringing the association between software evolution, concerns, agglomerations, and transverse dimensions into focus. We have made another contribution by stratifying the agglomerations as similarities and dissimilarities.

## C.6 CONCLUSION

After we defined the activities of our method, we looked forward to evaluate its applicability in answering some research questions, as the two ones that we presented in Section C.1.4. The questions seek to determine which strategy is more adequate when it comes to select software projects to extract concerns from. As we wanted to evaluate two possibilities, grouping systems according to their types or domains (or transverse dimensions as we call them – Section C.1.1), we agglomerated the association between source code artifacts and concerns as cases of full and partial similarities (as explained in Section C.1.2).

Combining transverse dimensions and the stratification of full and partial similarities is beneficial. Grouping software projects under a chosen transverse dimension can reveal the similitude of systems regarding their concerns. We trust that this is a better strategy compared to lacking of criteria in selecting and joining projects to inspect their evolution. As a result, we focused the studies discussed in Chapters 4 and 5 on the mining of concerns from the source code of systems selected from a particular domain of software: non-relational databases.



## CONCERNS IDENTIFIED DURING STUDY III

Concern	Purpose	Found in...(*)				
		D	S	M	G	T
Service-Orientation	allows communication with services or enables a system to provide services	X	X	X	X	X
Data Compression	supports data compression	X			X	
Configuration	performs automatic configuration of systems' modules and functionalities	X	X		X	
Web App Support	enables embedding of service-client protocols	X	X	X	X	X
Graph Computing	supports mathematical processing of graphs	X			X	X
Text Indexing	processes data in form of text		X		X	
Geospatial Processing	supports processing of geospatial data	X			X	
Test	automates self-tests of systems' modules	X	X	X	X	X
I/O Processing	accesses/processes information obtained from I/O devices				X	
Logging	allows logging of routines executed by programs	X	X	X	X	X
Data Format Processing	imports-exports from-to data formats, <i>e.g.</i> , xml, json	X	X	X	X	X
Process Execution	executes external processes, <i>e.g.</i> , OS programs	X	X	X		X
Database	enables the communication between client applications and databases	X	X	X	X	X

(\*) D – Distributed, S – Services, M – Mobile, G – Graph, T – Time Series

Concern	Purpose	Found in...(*)				
		D	S	M	G	T
Metrics and Measurement	measures systems' metrical/quality attributes	X			X	X
Stream Processing	enables data streaming	X		X		
Report	enables preview and printing of reports			X		
Serialization	supports serialization of data	X	X	X	X	X
Benchmark	enables benchmark tests in applications	X			X	
Distributed Computing	adds distributed-computing features to systems	X			X	X
ElasticSearch Processing	supports processing of document-based information		X			
Parsing	makes parsing and compiling of source code possible during runtime	X	X	X		X
Dataset Processing	enables processing of dataset formats, <i>e.g.</i> , CSV	X	X			
Security	adds features related to data security, <i>e.g.</i> , encryption	X	X	X	X	X
Dependency Injection	makes injection of components during runtime possible (hotplug)					X
Caching	supports defining caching strategies		X	X		
Encryption	enables data encryption		X	X	X	
Tracing	allows processing of tracing stacks			X		
Geometry	adds functionalities to process/calculate geometric shapes				X	
Bulding/Deploy	supports building and deploying programs' releases		X			
Authentication	enables authentication of users or client applications		X			
Cloud Computing	allows communication with cloud services			X		
Mailing	enables sending/receiving of electronic messages		X			
Messaging	implements message queues	X	X			
Validation	automates validation of data structures during runtime, <i>e.g.</i> , j-beans validation					X
Programming Utilities	provides special data structures, <i>e.g.</i> , lists	X	X	X	X	X
Barcode Reading	allows reading of barcodes			X		
Multimedia	enables integration with multimedia resources			X		
Bioinformatics	allows processing of biological data		X			
UI	allows creation of user interfaces	X		X	X	X
Cluster Management	manages clustered data	X				
Mathematical Processing	provides support for complex mathematical calculations	X			X	X

(\*) D – Distributed, S – Services, M – Mobile, G – Graph, T – Time Series

## STUDY IV – INSTANTIATING OUR METHOD UNDER A DIFFERENT DEVELOPMENT CONTEXTS

Our method is not limited to a specific type of programming language and development platform. As we explained in Section 3.1.3 (Figure 3.3), we designed it as a stack of interrelated activities. We also predicted that it is possible to instantiate our method to meet different contexts of development. This study has the intention to showcase this feature.

Javascript is the most popular language in github and npm<sup>1</sup> (hereafter, NPMRegistry) is the most popular online registry for javascript packages (KAVALER et al., 2019)(BORGES et al., 2015). The packages enclose third-party components and npm is responsible for distributing them and making information about them available. Javascript developers benefit from npm packages when they need to automate the injection of components in their applications. We see this as an opportunity to demonstrate the capability of our method to analyze javascript applications' concerns.

The main goal of this study is to expand the applicability of our method and AKS to support a different context of software development. Specifically, we have the intention to mine and visualize concerns of npm-enabled javascript applications (hereafter, npmjs systems/applications/projects).

Conducting this study requires: (i) automating our method's abstraction to allow the mining of npmjs applications; (ii) mining concerns from npmjs systems grouped under two domains: online chats and Role-Playing Games (RPG); (iii) visualizing the concerns mined from each domain.

We were able to use our method to identify npmjs systems' concerns. We also provide a preliminary analysis of how javascript applications' concerns agglomerate around the two aforementioned domains.

---

<sup>1</sup><https://www.npmjs.com/>

## E.1 MINING CONCERNS FROM NPMJS SOFTWARE PROJECTS

Similar to java projects, npmjs developers have devised a way to automate the injection of third-party components. They usually fill information about components as dependencies in package.json files. This means: package.json files have the same responsibility of java's POM and Gradle files. Listing E.1 shows some lines extracted from a package.json file. Developers of an online chat application, Rocket.Chat, created a package.json to automate the embedding of a set of components (from line 11 to 17, highlighted in yellow). As each component focuses on fulfilling a specific purpose, we consider that they are suitable for revealing developers' intentions regarding the implementation of concerns.

**Listing E.1** Example of a package.json file

```

1  {
2  "name": "Rocket.Chat",
3  "description": "The Ultimate Open Source WebChat Platform",
4  "version": "2.0.0-develop",
5  "author": {
6    "name": "Rocket.Chat",
7    "url": "https://rocket.chat/"
8  },
9  ...
10 "dependencies": {
11   "apollo-server-express": "^1.3.6" ,
12   "archiver": "^3.0.0" ,
13   "arraybuffer-to-string": "^1.0.2" ,
14   "atlassian-crowd": "^0.5.0" ,
15   "autolinker": "^1.8.1" ,
16   "aws-sdk": "^2.368.0" ,
17   "bad-words": "^3.0.2" ,
18   ...
19 }
20 ...
21 }
```

Package.json files are crucial during many deployment phases because they automate the downloading and configuration of components in the machines and environments where javascript systems must run. Consequently, developers pay very close attention to the names and versions of components that they add to package.json files. So, given their importance, package.json files are often part of javascript projects and the information that they contain tends to be correct and precise.

The name of components is a unique id which can be used to unambiguously identify them. Figure E.1 illustrates how our method uses components' names to spot concerns in npmjs applications.

Package.json files of two distinct software projects determine that they are dependant on a component called **moment**. We have empowered AKS with routines to retrieve **moment**'s metadata from NPMRegistry. Navigating to <https://www.npmjs.com/package/moment> reveals a collection of keywords: "moment", "datetime", "parse", "format", and



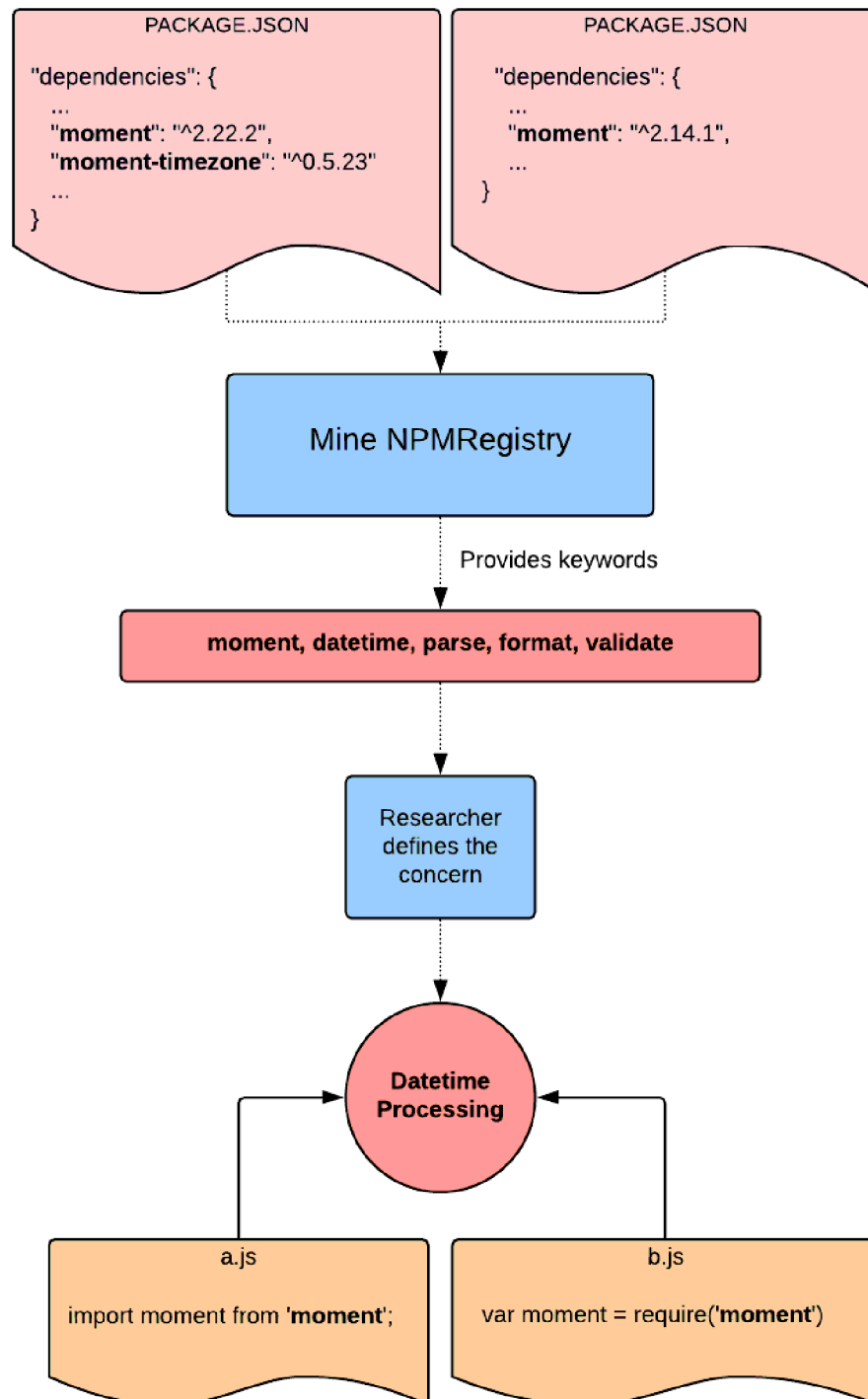


Figure E.1 Mining of Javascript Systems' Concerns

“validate”. AKS is capable of extracting keywords from NPMRegistry’s metadata. As NPMRegistry does not offer a direct information about any specific concern that can be associated with **moment**, our method relies on the keywords to support the manual def-

initiation of concerns. For instance, considering the aforementioned keywords, we associate **moment** with “Datetime Processing”.

The last step of our mining strategy requires finding npmjs applications’ source code artifacts that import **moment**. In Figure E.1, we exemplify two possible ways by which **moment** can be integrated into javascript artifacts: (i) using an import directive (as in “a.js” file), and (ii) adding a require directive (as in “b.js” file).

## E.2 STUDY DEFINITION

We rely on one of our previous findings to determine how to select and group npmjs software projects: domains of software are a more adequate way to group systems (as discussed in Chapter 4). Table E.1 describes the javascript applications that we chose to analyze. We gathered them from two domains: online chats and RPG. One important remark about the projects: they are npm-based and make use of package.json files to store information about components.

**Table E.1** Analyzed Projects

Domain	Project	Description
Chat	Lets-Chat <sup>a</sup>	A chat app for small teams
	Rocket-Chat <sup>b</sup>	An open source webchat platform
RPG	Adventurer’s Codex <sup>c</sup>	An open source tool for tabletop RPG
	Delaford <sup>d</sup>	A javascript 2D medieval RPG

<sup>a</sup> <https://github.com/sdelements/lets-chat>

<sup>b</sup> <https://github.com/RocketChat>

<sup>c</sup> <https://github.com/adventurerscodex>

<sup>d</sup> <https://github.com/delaford/game>

Compared to the previous studies, this one has some limitations. First of all, we do not examine the evolution of concerns. Package.json files are often stored in Version Control Systems (VCS). Therefore, as POM/Gradle files, they can enable the tracking of information about components (and concerns) through the evolution of npmjs software projects. Unfortunately, we have not added the capability of mining multiple versions of npmjs systems to AKS yet. As a consequence, the analysis described in this chapter refers to one single snapshot obtained from the latest versions of the applications.

The results that we present in the next section are based on a rudimentary strategy that we used to associate concerns with npmjs applications’ artifacts: being “C” a concern, “C” is linked to artifact “A” (a .js file) if “A” imports components which implement “C”. Moreover, the results do not address the Dedication to Concern (DtC) between “A” and “C”.

As in Studies I (Chapter 4), II (Chapter 5) and III (Appendix C), we rely on visualizations to exhibit our findings. The visualizations shown in Section E.3 use the Number Of Artifacts (NOA) and Lines Of Code (LOC) metrics. NOA refers to the number of distinct javascript artifacts that import a component associated with each concern. We use LOC to count the number of lines of code of the associated artifacts.

### E.3 RESULTS

The word-clouds exhibited in Figures E.2 and E.3 represent the union of concerns gathered from the two aforementioned domains (the complete list of concerns and their descriptions can be found in Appendix F). Both figures use the NOA metric to differentiate the concerns.

Figure E.2 exhibits chat applications’ cloud of concerns. The cloud reveals that some concerns are implemented by more artifacts than others: “Javascript Programming Support”, “String Processing”, and “Notification”.



**Figure E.2** Chat Applications’ Concerns Cloud

Figure E.3 presents the concerns that AKS mined from the RPG domain. We highlight “Front-end Development”, “Mobile Programming”, “Hashing”, and “Data Format Processing” as the concerns that are associated with a significant number of source code artifacts.



**Figure E.3** RPG Applications' Concerns Cloud

## E.4 DISCUSSION

We briefly explored the use of our method and tool to address a different software development context. We replaced java software projects with javascript applications to demonstrate one of our method's features which we described in Section 3.1: enabling different realizations and instances of its abstraction.

Javascript is a programming language that has received considerable attention from developers. It has been used in the making of many modern applications and it counts on an extensive number of third-party components to support the implementation of concerns. This motivated us to enhance AKS and reproduce the mining and analysis steps which we applied to investigate java-based software projects to examine npmjs applications too. As a result, we ended up with a preliminary view on the concerns that javascript developers embedded in four npmjs systems.

As future work, we must reshape this preliminary study as a full study, as the ones we presented in Chapters 4 and 5. However, we must enhance our method and AKS to parse and analyze javascript applications' source code elements more properly. We also think it is highly desirable to measure the DtC between concerns and artifacts of npmjs systems.

It is possible to replicate this study to address another relevant programming language:

python<sup>2</sup>. According to stackoverflow’s 2019 developers survey<sup>3</sup>, Python is the second most preferred programming language and it is the top one that developers want to learn. As python has become a developers’ favorite, it is important to support the analysis of concerns embedded in this type of application.

Python developers also rely on third-party components as a way to reduce effort while implementing applications. Like java and javascript developers, they usually indicate the components that must be downloaded and integrated into their programs. For instance, developers of SSBio, a bioinformatics framework written in python, added a requirements.txt file to its codebase<sup>4</sup>. The file lists all components that SSBio depends on. Listing E.2 displays some lines extracted from SSBio’s requirements.txt.

**Listing E.2** Example of a requirements.txt file

```
1 ...
2 requests >=2.13.0
3 cobra >=0.6.1
4 biopython >=1.69
5 bioservices
6 xmldict
7 mmtf-python
8 msgpack-python
9 awesome-slugify
10 ipywidgets==7.0.0
11 lxml
12 json-tricks >3.0
13 ...
```

Each line exhibited in Listing E.2 corresponds to a distinct third-party component. Same as using MVNRepository to retrieve metadata about java-based components and NPMRegistry to gather metadata about npmjs components, python developers rely on the Python Package Index<sup>5</sup> (PyPi) to obtain details about python components. For instance, navigating to <https://pypi.org/project/biopython/> brings metadata about one component mentioned in Listing E.2: **biopython** (line 4). The metadata contains a list of topics associated with **biopython**: “Scientific/Engineering”, “Scientific/Engineering :: Bio-Informatics”, and “Software Development :: Libraries :: Python Modules”. Considering the topics, it seems logical that “Bioinformatic” is an adequate concern to represent **biopython**. Table E.2 displays other concerns that we freely and manually defined based on the content of SSBio’s requirements.txt file and the information found in PyPi.

PHP<sup>6</sup> developers add a composer.json file to their software projects as a way to incorporate third-party components. Listing E.3 exhibits some lines from the composer.json of Sylius<sup>7</sup>, an open source e-commerce platform. Packagist<sup>8</sup> is an online platform that developers can consult to know more about PHP components. For instance, they can retrieve

---

<sup>2</sup><https://www.python.org/>

<sup>3</sup><https://insights.stackoverflow.com/survey/2019>

<sup>4</sup><https://github.com/SBRG/ssbio>

<sup>5</sup><https://pypi.org>

<sup>6</sup><https://www.php.net/>

<sup>7</sup><https://github.com/Sylius>

<sup>8</sup><https://packagist.org/>

**Table E.2** SSBio’s Concerns

Component	Description	Concern
requests	http library	Web App Support
cobra	genome-scale modeling	Bioinformatics
bioservices	bioinformatics web services	Service-Orientation <sup>a</sup>
xmldict	XML processing library	XML Creation/Parsing
mmtf-python	MacroMolecular Transmission Format (MMTF) encoder	Biological Structures Encoding <sup>b</sup>
msgpack-python	binary serialization	Serialization
awesome-slugify	URL slugify <sup>c</sup> library	URL Processing
ipywidgets	HTML widgets package	UI
lxml	XML processing library	XML Creation/Parsing
json-tricks	extra features for JSON	JSON Creation/Parsing

<sup>a</sup> Perhaps, a more adequate decision would have to consider overlapping two concerns to represent **bioservices**: “Service-Orientation” and “Bioinformatics”

<sup>b</sup> Naming this concern with precision would require the help of a bioinformatics specialist

<sup>c</sup> “Slugify” is a process by which friendly URLs are generated from lengthy ones

details about **egulias/email-validator** (line 3) by browsing the component’s packagist’s webpage: <https://packagist.org/packages/egulias/email-validator>. As a result, Packagist returns its description and keywords, which can be used to associate a concern with the component. The keywords “email”, “validator”, “validation”, “emailvalidation”, and “emailvalidator” indicate that either “Validation” or “Email Validation” is a suitable concern.

**Listing E.3** Example of a composer.json file

```

1  "require": {
2      ...
3      "egulias/email-validator": "~2.0",
4      ...
5      "fzaninotto/faker": "^1.6",
6      "gedmo/doctrine-extensions": "^2.4.12",
7      "jms/serializer-bundle": "^2.0",
8      ...
9      "liip/imaginary-bundle": "^2.0",
10     "payum/payum": "^1.4",
11     ...

```

Based on the information fetched from Packagist, we determined a set of concerns to represent the components shown in Listing E.3. In Table E.3 we summarize the association between the chosen concerns and Sylius’ components.

Ruby<sup>9</sup> is another programming language that developers combine with third-party components to create applications. Adding a Gemfile to software projects is a good practice that ruby developers follow. Gemfiles have the same responsibility of POM, Gradle,

<sup>9</sup><https://www.ruby-lang.org/en/>

**Table E.3** Sylius's Concerns

Component	Description	Concern
egulias/email-validator	library for validating emails	Validation
fzaninotto/faker	library that generates fake data	Mocking
gedmo/doctrine-extensions	extensions for Doctrine Object-Relational Mapping (ORM)	ORM Processing
jms/serializer-bundle	library for serializing and deserializing data	Serialization
liip/Imagine-bundle	image manipulation abstraction toolkit	Image Processing
payum/payum	integration with more than 50 payment services	Payment

package.json, composer.json, and requirements.txt files. They are meant to comprise information about the injection of components in ruby projects. PopHealth<sup>10</sup>, an open source population health reporting application written in ruby, has a Gemfile. Listing E.4 shows some of its lines.

**Listing E.4** Example of a Gemfile

```

1 source 'http://rubygems.org'
2
3 gem 'rails', '~> 4.1.2'
4 gem 'quality-measure-engine', '3.1.0'
5 gem "hqmf2js", :git=> "https://github.com/pophealth/hqmf2js.git"
6 gem 'health-data-standards', '3.5.0'
7 gem 'nokogiri'
8 gem 'rubyzip'
9 ...

```

Ruby developers also count on a website to achieve information about components: rubygems<sup>11</sup>. With its help we associated the concerns shown in Table E.4 with the components listed in Listing E.4.

The concerns described in Tables E.2, E.3, and E.4 are not precise because defining them adequately demands a thorough examination. So, it is necessary to determine how to measure DtC according to the characteristics of each mentioned programming language. Re-running our action research study while improving AKS to deal with a multitude of languages and their respective components-related artifacts and websites is necessary.

## E.5 CONCLUSION

One key aspect of our method resides in the way how we split it into interrelated activities. We noticed that its abstraction (described in Section 3.1.1) could be instantiated

<sup>10</sup><https://github.com/pophealth/popHealth>

<sup>11</sup><https://rubygems.org/>

**Table E.4** PopHealth’s Concerns

<b>Component</b>	<b>Description</b>	<b>Concern</b>
rails	fullstack web framework	Web App Support
quality-measure-engine	library for clinical quality measurement	Metrics and Measurement
hqmf2js	Health Quality Measure Format (HQMF) conversion library	Health Data Processing <sup>a</sup>
health-data-standards	healthcare-related formats processing library	Health Data Processing <sup>b</sup>
nokogiri	HTML, XML, SAX, and Reader parser	Data Formats Processing
rubyzip	module for reading/writing zip files	Compression

<sup>a,b</sup> Defining an adequate concern would require the help of healthcare specialists

under varied contexts of software development and concerns analyses. In this chapter we exemplified different types of applications and programming languages that have made the use of third-party components to implement concerns. It is also usual that developers add information about their components in projects’ metadata artifacts. Consequently, our method can take advantage of such widespread practice to spot concerns regardless adopted development technologies.



## CONCERNS IDENTIFIED DURING STUDY IV

Concern	Purpose	Found in...(*)			
		L	R	A	D
Datetime Processing	parses and formats date and time data	X	X		X
Javascript Programming Support	provides basic javascript data types, structures, and functionalities	X	X	X	X
Database	allows communication with databases	X	X		
Authentication	supports validation of users' credentials	X	X		
Asynchronous Programming	enables execution of asynchronous processes	X			
Cookie Processing	allows inserting/retrieving data into/from cookies	X			
Socket-Based Communication	supports socket-based communication between modules and processes	X			X
XMPP Processing	allows applying XMPP (a standard for messaging and presence)	X			
File and Directory Processing	supports manipulation of files, file types, and directories	X	X	X	
User Input Processing	enables collecting and parsing users' input	X	X		
Express Programming Support	allows development of web applications	X	X		X
YAML Creation/Parsing	provides ways to create and parse YAML files	X			

(\*) L – Lets.Chat, R – Rocket.Chat, A – Adventurer's Codex, D – Delaford

Concern	Purpose	Found in...(*)			
		L	R	A	D
Callback Processing	allows implementing and responding-to callbacks	X			
Encryption	enables encryption of data	X	X	X	
Hashing	generates and processes hashed data	X		X	
Templating	enables creating, parsing, and validating data templates	X			
Node Programming Support	allows development of node-oriented applications	X	X	X	X
Security	supports the protection of systems' data and functionalities	X			X
Data Format Processing	imports-exports from-to data formats, <i>e.g.</i> , xml, json	X		X	
Compression	supports data compression	X	X	X	X
Process Execution	executes external/internal processes	X			
Test	automates testing of routines		X		X
Chat	supports development of chat applications		X		
Cloud Computing	allows communication with cloud services		X		
String Processing	enables encoding/decoding and parsing of string-based data		X		
Image Processing	supports processing of image files		X	X	
Version Control	allows integration with VCS		X		
Front-end Development	enables integration of front-end frameworks, <i>e.g.</i> , React, Vuejs		X	X	X
Service-Orientation	allows communication with services or enables a system to provide services		X		
Data Format Processing	imports-exports from-to data formats, <i>e.g.</i> , xml, json		X		
Notification	automates notification routines		X	X	
CSS Creation/Parsing/ Transformation	automates processing of front-end's stylesheet		X	X	
File Storage	manages storage of data files		X		
Offline Storage	manages offline storage of data		X		
Clipboard	allows integration with OS clipboard		X	X	
System Specs Parser	support parsing of frameworks' and programs' specifications		X		
HTML Creation/Parsing	enables creating and parsing HTML during runtime		X	X	
Visualization	allows visualization of data		X		
File Type Processing	supports processing of file types		X		
URL Processing	automates formatting and processing URLs		X	X	
Stream Processing	enables data streaming		X		
SMS Processing	enables sending/receiving SMS		X		
Emojis	supports visualization of emojis		X		X

(\*) L – Lets.Chat, R – Rocket.Chat, A – Adventurer's Codex, D – Delaford

Concern	Purpose	Found in...(*)			
		L	R	A	D
Mathematical Processing	provides support for complex		X	X	
Caching	allows definition of caching strategies		X		
Metrics and Measurement	measures systems' metrical/quality attributes		X		
Binary Data Processing	processes binary/byte-coded data		X		
Blockchain Processing	supports processing of blockchain structures		X		
Pretty Printing	allows pretty-printing formatting of data		X		
PDF processing	enables creating and exhibiting PDF documents		X		
QRCode Processing	automates recognition of QRcodes		X		
Mail Processing	enables sending/receiving of electronic messages		X		
Bugs Processing	automates management of bugs		X		
Social Media Integration	allows integration with social media platforms		X		
LDAP Processing	enables integration with LDAP servers		X		
Logging	allows logging of routines executed by programs		X		
Charset Encoding	automates encoding/decoding of charsets		X		
Java Parsing/Compiling	enables parsing and compiling javascript code		X	X	
Validation	automates validation of data structures, <i>e.g.</i> , json		X	X	
Messaging	allows sending/receiving text messages		X	X	
Bundling and Deployment	automates creation and deployment of executable packages			X	
Web Gaming	supports development of online web games			X	
Mobile Programming	adds mobile features to web applications			X	

(\*) L – Lets.Chat, R – Rocket.Chat, A – Adventurer's Codex, D – Delaford



## DATASETS' SPREADSHEETS FORMAT

All tables contained in this Appendix follow the same pattern: *Column* indicates the name of the columns that can be found in our studies' spreadsheets. *Description* describes the purpose of each column. We exemplify a value of the column in *Example*.

### G.1 STUDY II'S SPREADSHEETS FORMAT

Column	Description	Example
Project	the name of the project that we analyzed	Neo4j
Dimension	project's domain	Graph Database
Reference	commit's hash of the snapshot under analysis	-
Label <sup>a</sup>	a label/name to represent the reference	Neo4j
Concern	a software concern	Service-Orientation
Analysis	mongodb's hash that identifies where AKS stored the data about the concern	-
Imports	components' import declarations that AKS associates with concerns	...jersey.api.core.HttpContext
File	the original source code file which AKS mined the concern from	.../neo4j/server/database/DatabaseProvider.java
Sourcecode <sup>b</sup>	path to a copy of the source code file	/aks/datasets/.../fdeb...java
LOC	value of the Lines of Code metric calculated from the source code file	43
NOC <sup>c</sup>	value of the Number of Classes found in the source code file	1
ICD <sup>d</sup>	value of the Imported Components' Dedication metric	0.5

Column	Description	Example
NOM <sup>d</sup>	value of the Number of Methods metric	2
NOR <sup>d</sup>	value of the Number of References metric	1
MD <sup>d</sup>	value of the Method's Dedication metric	0.5
DTC <sup>d</sup>	value of the Dedication to Concern metric	moderate

<sup>a</sup> In our scripts, we used the labels instead hashes, because the labels are more intuitive

<sup>b</sup> We add the copies to our replication packages as a way to provide other researchers with a direct and easy access to the source code files that AKS processes

<sup>c</sup> Although we do not use this metric, we are making it available for future work

<sup>d</sup> We described this metric in Section 3.1.4.1 (Table 3.1)

## G.2 STUDY III'S SPREADSHEETS' FORMAT

Column	Description	Example
Project	the name of the project that we analyzed	Neo4j
Dimension	project's domain	Graph Database
Reference	commit's hash of the snapshot under analysis	-
Label <sup>a</sup>	a label/name to represent the reference	Neo4j
Concern	a software concern	Service-Orientation
Analysis	mongodb's hash that identifies where AKS stored the data about the concern	-
Imports	components' import declarations that AKS associates with concerns	...jersey.api.core.HttpContext
File	the original source code file which AKS mined the concern from	.../neo4j/server/database/DatabaseProvider.java
Sourcecode <sup>b</sup>	path to a copy of the source code file	/aks/datasets/.../fdeb...java
AMW <sup>c</sup>	value of the Average Method Weight metric (LANZA; MARINESCU, 2010)	1
WMC <sup>d</sup>	value of the Weighted Methods per Class metric	4

<sup>a</sup> In our scripts, we used the labels instead hashes, because the labels are more intuitive

<sup>b</sup> We add the copies to our replication packages as a way to provide other researchers with a direct and easy access to the source code files that AKS processed

<sup>c</sup> Although we do not use this metric, we are making it available for future work

<sup>d</sup> We described our use of this metric in Section C.1.3

**G.3 STUDY IV'S SPREADSHEETS' FORMAT**

<b>Column</b>	<b>Description</b>	<b>Example</b>
Project	the name of the project that we analyzed	RocketChat
Dimension	project's domain	Chat
Concern	a software concern	Cloud Computing
Imports	components' import declarations	google-cloud/storage
File	the original source code file which AKS mined the concern from	.../app/google-vision/server/googlevision.js
LOC	value of the Lines of Code metric calculated from the source code file	163