Journal of Software Engineering Research and Development, 2024, 12:1, doi: 10.5753/jserd.2024.3086
This work is licensed under a Creative Commons Attribution 4.0 International License..

Using Third-Party Components' Metadata to Analyze Cross-cutting Concerns

Luis Paulo da Silva Carvalho [**Instituto Federal da Bahia** | ***luiscarvalho@ifba.edu.br***]
Thiago Souto Mendes [**Instituto Federal da Bahia** | ***thiagosouto@ifba.edu.br***]
Felipe Gustavo de Souza Gomes [**Universidade Federal da Bahia** | ***felipegustavo@dcc.ufba.br***]
Sávio Freire [**Instituto Federal do Ceará** | ***savio.freire@ifce.edu.br***]
Renato Lima Novais [**Instituto Federal da Bahia** | ***renato@ifba.edu.br***]
Manoel Gomes Mendonça [**Universidade Federal da Bahia** | ***manoel.mendonca@ufba.br***]

Abstract Context: Modularity is a key concept in software development. Well-modularized systems are easier to maintain and evolve, but achieving good modularity is difficult. Concerns that are important, but not central to a systems' main business rules, frequently end up scattered and entangled throughout several software modules. Those so called cross-cutting concerns are a major source of loss of modularity and code decay in software systems. **Motivation**: Studies on cross-cutting concerns often resort to manual identification of concerns, but manual identification is effort demanding, does not scale, and tends to be imprecise. Automatic approaches are therefore very attractive when the codebase is extensive. In modern systems, developers implement modules to address central business rules, but they tend to add third-party components in the codebase to materialize concerns related to other secondary aspects. Logging, database access, and tests automation are examples of concerns that are usually implemented with the help of imported components and are prone to scatter and tangle throughout the codebase. **Aims**: This paper proposes a method to track this type of cross-cutting concern. Our work takes advantage of the addition of metadata about components to track them. The method scales by automating the identification and analysis of concerns scattered throughout the software codebase. We define a new metric, Dedication to Concern (DtC), to measure how much source code modules focus on implementing the identified concerns. **Working Method**: We describe our method to mine cross-cutting concerns from the metadata related to the use of components. The method is instantiated as a tool, architectural knowledge suite (AKS). The tool is used to analyze concerns in a set of large Java projects. The results are used to feed an action research study, during which software development specialists analyze the AKS outputs to evaluate and evolve the method. **Conclusion**: The semi-automated approach is feasible and scalable, and can be used to analyze secondary concerns that are currently being imported into modern software systems via third-party components.

Keywords: *Mining, Concerns, Components, Static Analysis*

1 Introduction

Software development approaches depend on modularity as a critical concept to create, maintain, and evolve systems. Ideally, a software module should encapsulate the behavior of a single concern, *i.e.*, anything that stakeholders consider as a conceptual unit (Robillard and 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 can ensure that the maintenance and evolution of each concern require the modification of a few parts of a software system. However, the concerns can be tangled or scattered making it difficult the software evolution activities.

The tangling and scattering of concerns through source code are phenomena that affect the modularity of systems. While tangling is a state where lines associated with different concerns are interwoven (Hannemann and Kiczales, 2001), scattering is related to situations in which the code of a single concern spreads throughout multiple units of a system's codebase (Juhár and Vokorokos, 2015). Concerns that tend to scatter and tangle are known as *cross-cutting concerns*. Code fragments that implement cross-cutting concerns are hard to maintain because developers usually need to locate the concerns through multiple modules (He and Ye, 2015), impacting essential tasks in software development, *e.g.*, requirements analysis that involves refining and separating concerns to understand their interrelationships (Bellomo et al., 2014).

Keeping track of concerns through development is a time and effort-consuming task. Depending on the size of the codebase and how impacting the scattering and tangling of concerns are, executing this task can be counter-productive. Hence, it is important to define methods to identify and analyze concerns. Available methods have comprised the use of: (i) manual identification/mapping of concerns; (ii) information stored in software documents, *e.g.*, software requirements documents (SRDs) (Rosenhainer, 2004) and software architecture documents (SADs) (Díaz-Pace et al., 2016) can be used to find concerns; and (iii) automation based on static and dynamic analyses of software projects (Dit et al., 2013; Bernardi et al., 2016). Static analysis refers to the analysis of syntactic models extracted from the source code, *e.g.* abstract syntax trees and symbol tables. Dynamic analysis refers to approaches that trace the execution of software programs to find concerns.

The aforementioned methods are relevant and can help to locate and evaluate concerns, but they often fall short of being precise and adequate under unfavorable circumstances.

For example, developers frequently fail to create and update 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. Automated analysis can yield promising approaches to concerns identification.

In related literature, studies have proposed automatic and semi-automatic ways of identifying and visualizing concerns (Robillard and Murphy, 2002; Porubän and Nosál, 2014; Juhár and Vokorokos, 2015; He and Ye, 2015; Shaikh and Lee, 2016; Nunez-Varela et al., 2017; Carvalho et al., 2018, 2020). For example, Juhár and Vokorokos (2015) investigated what level of granularity is useful for most developers when dealing with concerns and Nunez-Varela et al. (2017) proposed a technique for finding relevant information from documents containing unstructured text. Despite current efforts, none of these studies investigated how third-party components' metadata can be used to identify concerns.

Third-party components are external modules that development communities make available for reuse. Developers usually implement modules to address core business rules, but they often import components in the source code to materialize concerns related to many secondary aspects (*e.g.*, logging, database access, security, and encryption). The use of third-party components has become common practice in modern software systems because importing components is a way to reduce development and maintenance effort (Agüero and Ballejos, 2017; Palyart et al., 2017).

The goal of this work is to take advantage of the metadata related to the use of components in software projects. We want to provide a way to extract and analyze information about cross-cutting concerns through the historical data of software systems. To achieve this goal, we performed the following activities:

1. We developed a method to identify cross-cutting concerns. The method comprises a set of activities that use metadata for the identification of third-party components contained in (public) software repositories and the use of those in the analyzed codebase (details in Section 2).
2. We automated our method as a tool, named architectural knowledge suite (AKS), to instantiate and automatize the activities of our method (described in Section 3).
3. We mined software repositories using AKS to produce a data set to allow the evaluation of our method (presented in Section 4).
4. We improved our method and tool by conducting an action research with the help of software development specialists (discussed in Section 6).

Although we have already used our method to conduct previous investigations (Carvalho et al., 2018, 2020), we have not center-pointed any study to evaluate it. In other words, we already applied our method as a part of other studies' strategy to extract concern-related information, but we have not dedicated any effort to pass it through the scrutiny of development specialists yet. Thus, we focus on hearing their opinions in this new study. After evaluating our method, our identification and classification of concerns were considered acceptable.

We have made AKS available for reuse, as it can help practitioners and researchers to perform further studies, together with the comprehensive data set produced for this study. The data set comprises: (i) a classification of concerns which we extracted from software projects; and (ii) a categorization of concerns regarding their relationship with systems' source code artifacts, as a metric which we call Dedication to Concern (DtC).

This paper is organized as follows: Section 2 presents our method. Section 3 shows how we created a tool, AKS, based on the method. Section 4 describes how we used AKS to fill concern-related information into a data set. In Section 6, we discuss the results of our action research study. We identify the threats to the validity of this work in Section 7. In Section 8, we elicit some related work. Our final remarks and future work can be found in Section 9.

2 Our Method

We developed a method to identify and analyze concerns in a semi-automatic manner. The method is composed of the activities, as shown in Figure 1.

We were able to automate all activities that Figure 1 identifies as "Automatic Activity". Few others, identified in the figure as "Manual Activity", are dependent on manual processing and data analysis. Activities 1 and 2 are dedicated to the mining of the concerns and the ones we now aim at to automate as much as possible. Activities 3 and 4 are dependent on the manual exploration of the data, but we consider them as part of the method as well, because the results of their analysis can be automated and incorporated to our method later on.

Although we focus on Java projects' throughout this work, the figure exhibits two possible ways to instantiate our method depending on the adopted development technology, *e.g.*, Java and JavaScript. We aim this study at Java because it was the first programming language that made us believe that it was possible to mine third-party components' concerns. This came from observing the preponderance of source code artifacts that could enable the extraction of such information, as we further explain while enumerating our method's activities:

1. Identify concerns: the method takes advantage of Java projects' POM and Gradle files and JavaScript systems' package files. Considering Java, mining components' metadata from POM/Gradle files[1] enables the recovering of developers' decisions related to the implementation of Java projects' concerns (activity 1.1). Depending on the chosen programming language, we complement the mining of the metadata by retrieving information about Java components from either MVNRepository[2] or JavaScript components from NPMRepository[3]. MVNRepository is a web portal responsible for indexing useful metadata concerning third-party Java components. Our method uses MVNRepository to

[1] Section 4 provides more details about POM/Gradle files
[2] https://mvnrepository.com/
[3] https://www.npmjs.com/

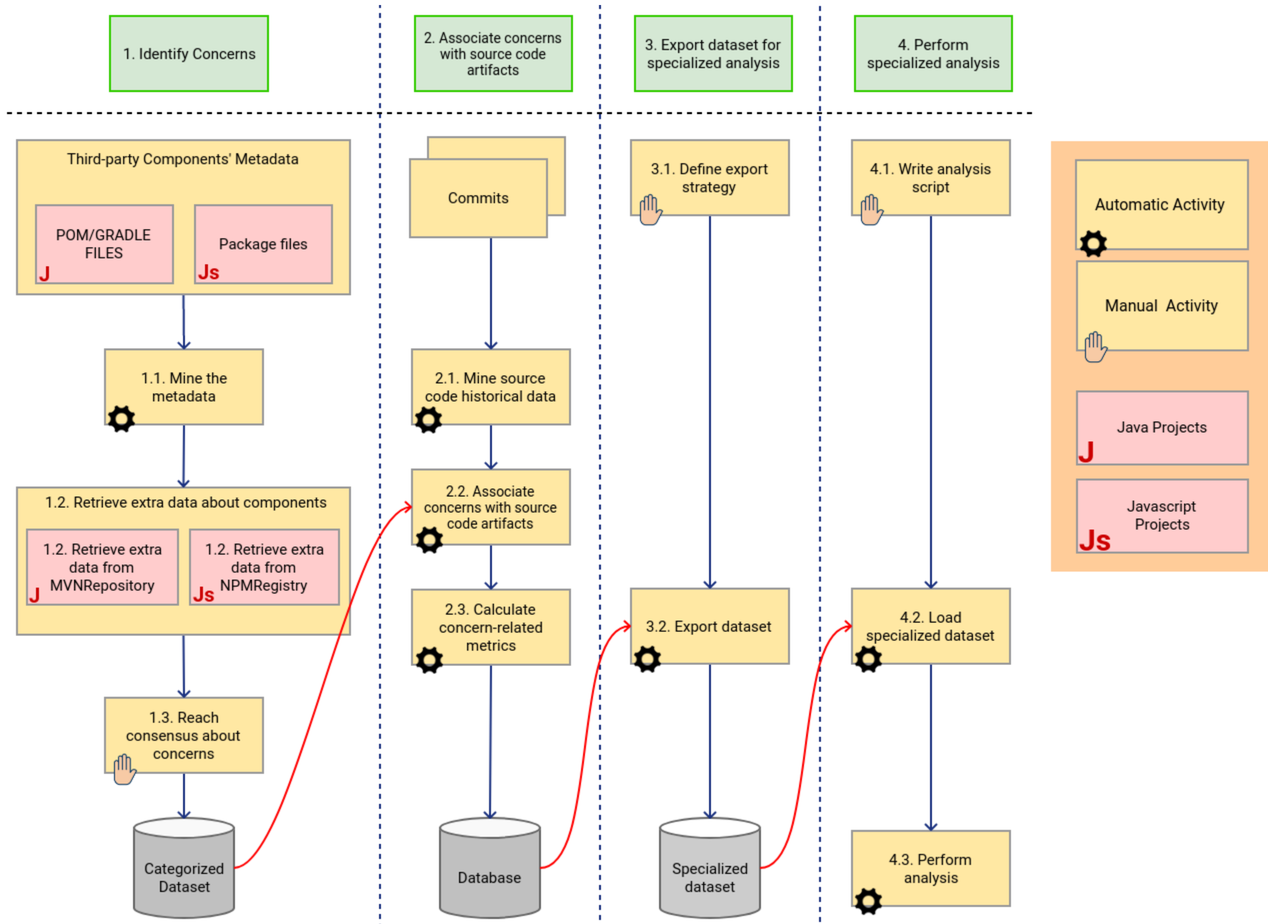

Figure 1. Our method's activities (Adapted from Carvalho et al. (2020))

mine categories for components found in POM/Gradle files (activity 1.2). As MVNRepository does not provide a category for all components[4], a manual classification of concerns is required (activity 1.3). The execution of this task generates a comprehensive data set of categorized concerns;

2. Associate concerns with source code artifacts: the main goal here is to determine which source code artifacts are affected by the concerns. For instance, in Java projects, this activity can associate the component, **dbunit**, with the ".java" files that import it (**import** *org.dbunit*) to automate tests. In addition, we calculate some important metrics related to the association between concerns and artifacts (more details in Section 4);

3. Export data set for specialized analysis: by mining concerns through the evolution of applications, our method has the potential to generate a large data set. We believe that it is advantageous to split the data into subsets to support specific studies. We achieve this goal by externalizing the information mined from software projects in a more concise reusable way: as a comma-separated-value (CSV) data set. We believe that exporting the data set as CSV files maximizes reusability because this type of file can be processed by different tools to automate varied investigations (*e.g.*, spreadsheet editors). 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 data set (activity 3.2);

4. Perform specialized analysis: we think that scripts are the best way to deal with the data mined from software projects. So, we have added activities to our method dedicated to write analysis scripts. We count on the use of R-language-based scripts to load (activity 4.2) and run the analysis (activity 4.3) on the data set generated by the previous activity (*export data set for specialized analysis*). Writing analysis routines as scripts (activity 4.1) favors the expansion of our approach to consider investigations other than the ones we have conducted (Carvalho et al., 2018, 2020).

As aforementioned we now focus on Java-based projects, but our method is generalized enough to embrace other programming technologies, *e.g.*, Javascript (as seen in Figure 1). We provide more information about the generalization of our method in Section 9.1.

3 Our tool: Architectural Knowledge Suite (AKS)

In this section, we describe our tool, named Architectural Knowledge Suite (AKS), that semi-automates our method. AKS allows us to mine concerns from software projects and determine how they are associated with source code artifacts. Figure 2 shows the main components of AKS' architecture. We grouped AKS features into four main modules: **miners**, **model**, **persistence**, and **exporters**.

[4] More information about why some components are not categorized by MVNRepository can be found in Raemaekers et al. (2017) and Velázquez-Rodríguez and De Roover (2020).

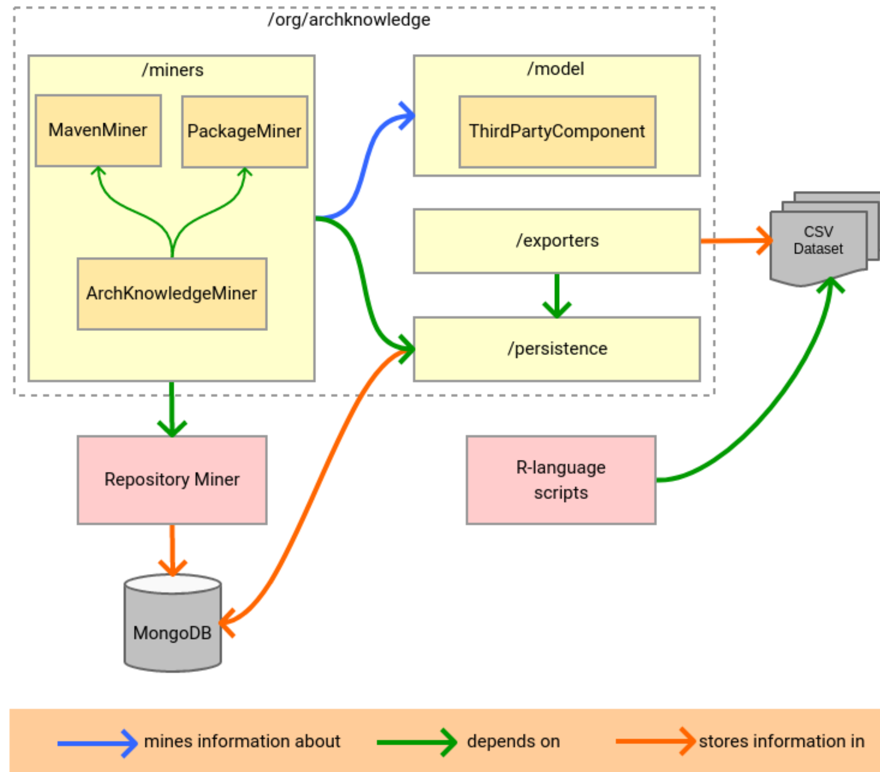

Figure 2. The Architecture of AKS

The **miners** module contains the mining strategies used by AKS to extract information about concerns from the source code of software projects. The module relies on the mining strategies made available by Repository Miner (RM) (Mendes et al., 2017).

Developed as an open source project using Java, RM enables the analysis of software repositories through the extraction and combination of data related to software evolution. RM has been mentioned and used by studies that seek to perform static analysis of source code and to investigate design problems (Mendes et al., 2015; Ibiapina et al., 2018; Gomes et al., 2019; Dias et al., 2019; Khomyakov et al., 2019; Mendes et al., 2019; de Freitas Farias et al., 2020). RM is distributed in the form of a JAR (Java ARchive) file and provides an API (Application Programming Interface), so that its features can be easily accessed through Java applications Mendes et al. (2015). Consequently, as an extension of RM, AKS is also Java-based. It can run independently to analyze the source code of software projects as command-line tool. We have not empowered it with a graphical interface yet, because, firstly, we want to stabilize its concerns mining and analysis strategies.

AKS encapsulates a set of generic routines to mine and process concerns-related information from third-party components. It relies on two sub-miners: (i) **MavenMiner** to mine concerns from Java-based projects, and (ii) **PackageMiner** when it is necessary to mine concerns from JavaScript systems.

Via RM, the miners store useful information about the static analysis software projects and their artifacts: a set of metrics related to components injection (more details in Table 1), our metric DtC (described in Section 4.1), and the association between DtC and software projects' artifacts. However, AKS counts on its own **persistence** module to store specific data about third-party components and concerns (represented by **ThirdPartyComponent** in the **model** module). In other words, AKS complements the information mined by RM with other ones related to the association between source code artifacts, components and concerns (more details in Section 4).

We have added custom routines to AKS **exporters** module to export the mined information to CVS data sets. As mentioned in Section 2 we trust that CSV files are a portable way to share data about the impact of concerns on software projects.

4 Mining Information about cross-cutting Concerns

Figure 3 contains an example that shows how AKS automates the mining of concerns from third-party components metadata (the first activity of our method).

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). We have focused on the metadata that developers use to inform which components they embed 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).

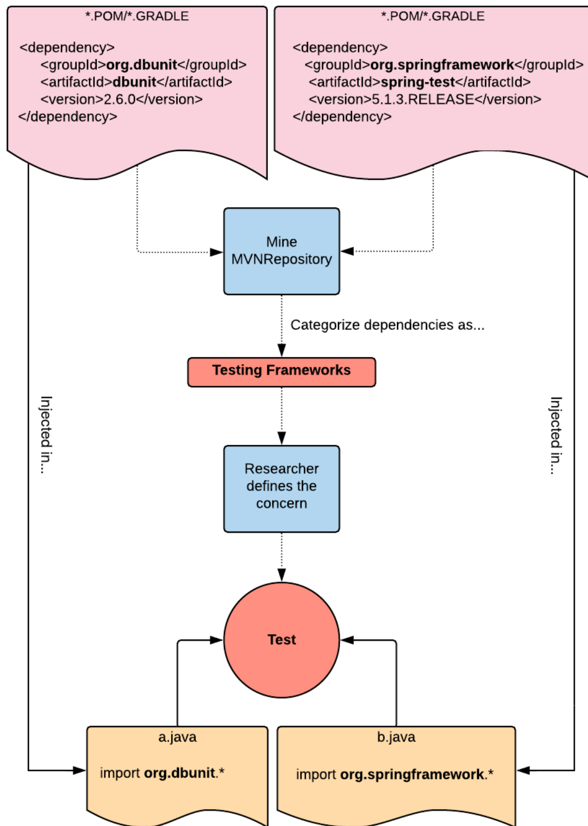

Figure 3. Mining of Concerns (Carvalho et al., 2020)

We are interested in Project Object Model (POM) and Gradle files that developers use to inject components in Java-based software projects Agüero and Ballejos (2017); Palyart et al. (2017). This comes the fact that such files have became plentiful because many Java developers have adopted them to encapsulate information about components used in their projects.

Suppose that developers added the components, **dbunit** and **spring-test**, to the POM file of a software project as exhibited in Listing 1. AKS is capable of processing the POM files to mine the components' IDs: groupId and artifactId. As the IDs ensure that each component is uniquely and unambiguously identified, AKS uses them to retrieve metadata from MVNRepository. The tool navigates to two distinct MVNRepository's pages obtained from the combination of components' group and artifact IDs: dbunit's metadata and spring-test's metadata.

Listing 1: Example of a POM file

```
1  <project>
2    ...
3    <dependencies>
4      <dependency>
5        <groupId>org.dbunit</groupId>
6        <artifactId>dbunit</artifactId>
7        <version>2.6.0</version>
8      </dependency>
9      <dependency>
10       <groupId>org.springframework</groupId>
11       <artifactId>spring-test</artifactId>
12       <version>5.1.3.RELEASE</version>
13     </dependency>
14   </dependencies>
15 </project>
```

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".

AKS associates source code artifacts with concerns by examining the list of import declarations in ".java" files. In Figure 3, file "a.java" imports **dbunit** (or we can say that **dbunit** is imported in "a.java") and "b.java" imports **spring-test** (developers imported **spring-test** in "b.java"). In this case, AKS associates the "Test" concern with both artifacts.

4.1 Measuring Dedication to Concern

Using import declarations to associate concerns with code units does not suffice for a deeper analysis of how concerns impact information systems. The main problem is: importing a component does not guarantee that it is extensively used by an artifact. Developers may avoid analyzing such situations or rank them down to a low-priority category of impact. To address this issue, we needed a set of metrics to measure how much the imported components are used by the source code's artifacts.

First, we consulted related literature to reuse metrics, but not all available ones suited us. At first glance, the metrics Number of Components (NOC) (Abilio et al., 2015; Nuñez-Varela et al., 2017), Program Element Contribution (CONT) (Eaddy et al., 2008), Concern Diffusion over Components (CDC) (Sant'Anna et al., 2003), and Concern Diffusion over Operations (CDO) (Sant'Anna et al., 2003) looked reusable, but further analyzing them, we realized that they did not entirely match our needs.

NOC, for instance, counts how many components (constants/refinements) are necessary to implement a feature (Abilio et al., 2016). We are not sure if we can apply both concepts, "concern" and "feature", interchangeably. The 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 artifacts that use/implement concerns. Additionally, considering the import declarations is a more direct reference to how much a software artifact is dedicated to either few concerns (via few imports or imports of components related to few concerns) or many concerns (by importing too many components associated with a varied plethora of concerns).

CDC metric counts the number of primary components whose main purpose is to contribute to the implementation of a concern (Sant'Anna et al., 2003), and CDO counts the number of primary operations whose main purpose is to contribute to implementing a concern (Sant'Anna et al., 2003). According to the definitions, respectively, the metrics accumulate the total number of components (classes or aspects) 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 during each 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 them. CDC and CDO take a concern "C" as input and return to the total number of components and operations that refer to "C".

So, CDC and CDO do not entirely fit in what we needed 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/selected the metrics described in Table 1.

With the purpose of measuring the relationship between artifacts and concerns, we defined the metric *Dedication to Concern* (DtC), as "the degree to which a source code artifact (*e.g.*, a .java file of a Java-based project) is dedicated to implementing a given concern". We applied our set of 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. Also, we propose measuring DtC against a three-factor qualitative scale because, 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 applied a simple percentage ratio (Dawson and O'neill, 2003) to calculate the metrics Imported Components' Dedication (ICD) and Methods' Dedication (MD). This enabled us to fit both metrics' numeric values into the three categories of DtC 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:

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

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 or HIGH, and **MD** is HIGH.

We adopted the following as the values 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 (Hensler and Stipak, 1979; Böhme and Freiling, 2008; Philippi, 2021). Additionally, interval scales can be converted to nominal scales by being cut at breakpoints and assigning the resulting slices of data to categories (Böhme and Freiling, 2008), such as the ones that we associated with DtC (SLIGHT, MODERATE, and HIGH).

5 Applying our DtC Metric - A Worked Example

Now, we exemplify how 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 2 describes the databases.

5.1 Measuring a High DtC

The first category of DtC, high, includes cases as the one exemplified in Listing 2. 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 cyan), **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 2: 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 newest
21     assertNull(cache.cacheItem("four")); // This
       should boot out 'two'
22     assertNull(cache.cacheItem("two")); // Should
        have booted 'three'
23     assertNotNull(cache.cacheItem("one"));
24     assertNull(cache.cacheItem("three")); //
       Should have booted 'four'
25     assertNotNull(cache.cacheItem("one"));
26   }
27
28   @Test
```

Table 1. Dedication to Concern's Metrics

Metric	Description	Purpose
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[a]	Number of Methods	Total number of methods of a source code artifact
NOR[b]	Number of References	NOR counts the total number of references to a component/concern found in methods
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 regarding 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[c]	Dedication to Concern	DtC outputs the dedication of a source code artifact from values obtained from ICD and MD

[a] from Oliveira et al. (2014); Nunez-Varela et al. (2017)
[b] 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
[c] similar to Concern Diffusion over Components (CDC) (Sant'Anna et al., 2003), except that our metric applies to only one class per measurement

Table 2. Non-relational Databases

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

[a] https://github.com/JanusGraph/janusgraph
[b] https://github.com/neo4j/neo4j
[c] https://github.com/kairosdb/kairosdb

```
29   public void test_uniqueCache() {
30     TestObject td1 = new TestObject("td1");
31     TestObject td2 = new TestObject("td2");
32     TestObject td3 = new TestObject("td3");
33
34     DataCache<TestObject> cache = new DataCache<
       TestObject>(10);
35
36     cache.cacheItem(td1);
37     cache.cacheItem(td2);
38     cache.cacheItem(td3);
39
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 describes how AKS calculates the metrics for the artifact exhibited in Listing 2 and determine its DtC considering the "Test" concern.

5.2 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 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: 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(RandomGenerator.class);
13
14   private static final int standardLower = 7;
15   private static final int standardUpper = 21;
16
17   public static String[] randomStrings(int number
       ) {
18     return randomStrings(number, standardLower,
       standardUpper);
19   }
20
21   public static String[] randomStrings(int number
       , int lowerLen, int upperLen) {
22     String[] ret = new String[number];
23     for (int i = 0; i < number; i++) ret[i] =
       randomString(lowerLen, upperLen);
24     return ret;
25   }
26
27   public static String randomString() {
28     return randomString(standardLower,
       standardUpper);
29   }
30
31   public static String randomString(int lowerLen,
       int upperLen) {
32     assert lowerLen > 0 && upperLen >= lowerLen;
33     int length = randomInt(lowerLen, upperLen);
34     StringBuilder s = new StringBuilder();
35     for (int i = 0; i < length; i++) {
36       s.append((char) randomInt(97, 120));
37     }
```

Table 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"

```
38      return s.toString();
39    }
40
41    public static int randomInt(int lower, int
        upper) {
42      assert upper > lower;
43      int interval = upper - lower;
44      // Generate a random int on [lower, upper)
45      double rand = Math.floor(Math.random() *
        interval) + lower;
46      // Shouldn't happen
47      if (rand >= upper) rand = upper - 1;
48      // Cast and return
49      return (int) rand;
50    }
51
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 4 demonstrates how AKS calculates the metrics to classify the source code artifact in Listing 3 as moderately dedicated to implement the "Test" concern.

5.3 Measuring a Slight DtC

Listing 4 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 imported from **junit**[5] (lines 65, 74, 81, and 93). Consequently, regarding the "Test" concern, we determine that the artifact is **slightly dedicate** to implement it.

Listing 4: 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;
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;
```

[5] 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.

Table 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"

```
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(storeDir).
     newGraphDatabase();
71   }
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(), populationResult.
     maxPropertyId + "");
99       Node nodeByStringProperty =
100          db.findNode(Label.label(LABEL),
     getStringProperty(), populationResult.
     maxPropertyId + "");
101      assertNotNull("Should find last inserted
     node", nodeByStringProperty);
102      assertEquals(
103          "Both nodes should be the same last
     inserted node",
104          nodeByStringProperty,
105          nodeByUniqueStringProperty);
106
107      Node nodeByUniqueLongProperty =
108          db.findNode(Label.label(LABEL),
     getUniqueLongProperty(), populationResult.
     maxPropertyId);
109      Node nodeByLongProperty =
110          db.findNode(Label.label(LABEL),
     getLongProperty(), populationResult.
     maxPropertyId);
111      assertNotNull("Should find last inserted
     node", nodeByLongProperty);
112      assertEquals(
113          "Both nodes should be the same last
     inserted node",
```

```
114             nodeByLongProperty,
115             nodeByUniqueLongProperty);
116         }
117     }
118
119     ...
120
121 }
```

In Table 5 we show how our method calculates the metrics to determine the dedication of the source code artifact exhibited in Listing 4. 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.

6 An Action Research Study to Evaluate and Refine our Method

In the previous sections, we provided details about how our method supports the identification and analysis of concerns. We now describe an action research study that we conducted to validate and enhance our method and tool. Action research studies stem from the principle of cyclical field intervention, which allows the testing and refinement of theories in practice (Baskerville, 1999; Puhakainen and Siponen, 2010). 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.

Dos Santos and Travassos (2011) defined a template to report action research studies. Their template comprises a set of activities: **Diagnosis**, **Planning**, **Actions**, **Evaluation and Analysis** and **Reflections and Learning**. The next sections describe the activities and how we instantiated them.

6.1 Diagnosis

Diagnosis is composed of three sub-phases: problem description (PD), research theme (RT), and project context (PC). PD describes the problem faced, and RT summarizes the study to limit its scope. In turn, PC informs where the problem happens. We summarize our PD 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 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

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

6.2 Planning

Planning (P) must include information about: the technical aspects and literature surveys to ground the study (P1), controlled/pilot studies to determine the risks of using software technologies (P2), and the operational elements that are necessary to execute the research (P3). Our research benefits from the theoretical and technical topics and resources about concerns mining and analysis found in previous work (Canfora and Cerulo, 2005; Sant'Anna et al., 2007; Bernardi et al., 2016; Marçal et al., 2016) (P1). Specifically, we focus on **static analysis** of source code artifacts which are retrieved from the **development history of software systems** (Canfora and Cerulo, 2005). We already applied our method and tool in two previous studies (Carvalho et al., 2018, 2020), so we are confident AKS is robust enough to support new investigations (P2) and can provide us with a precise data set to interact with software development specialists (P3).

6.3 Actions

Actions are concerned with putting the study's tasks and interventions in chronological order. Figure 4 shows how we organized our study's actions.

Analyzing the figure, the first action comprises all activities that we performed to prepare the study (1), *e.g.*, producing an agreement data set (more details about our agreement data set in Section 6.3.1 ahead). We sent the data set to the specialists after instructing them about how to manipulate it. After they returned their tagged data sets (2), we processed them through Kappa's Coefficient to calculate the strength of agreement/disagreement (3). The Cohen's Kappa Coefficient (hereafter, Kappa) applies to situations in which it is necessary to use descriptive statistics to summarize the agreement between two assigners (or judges, or raters) across several objects (*e.g.*, things, statements, calculations) (Cohen, 1960; Brennan and Prediger, 1981). Kappa can calculate the proportion of the agreement while correcting it for chance (Cohen, 1960). In case we reached no agreement, we took the opportunity to collect specialists' opinions about why they disagreed. We then used the 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.

By measuring Kappa, we can reach any of the strengths of the agreement listed in Table 6. Advancing through the categories exhibited in the Table (from <0.00 up to 1.00) means that raters agree about the same items. We consider that a poor or slight agreement is undesirable because it means that the raters disagree about the majority of the information in

Table 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

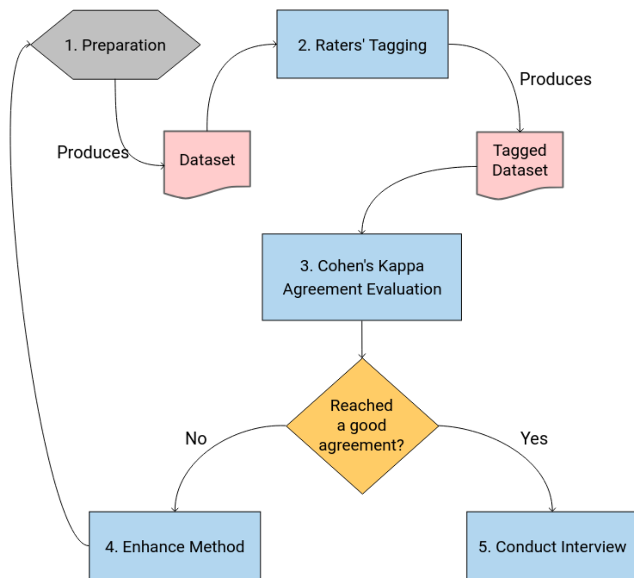

Figure 4. Actions of our Action Research Study

the data set. We aimed at reaching a strength beyond "Fair" to end the evaluation of our method.

Table 6. Strength of Agreement (Landis and 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

6.3.1 Generating the Study's Agreement Data Set

As aforementioned, the preparation of our study (Action 1 in Figure 4) has the purpose of creating an agreement data set so that software development specialists could help us to evaluate our method. Therefore, we configured and ran AKS to mine concerns from projects of a particular domain of software: non-relational databases. Choosing "domain of software" as a strategy to select software projects is a consequence of one of our previous studies' findings (Carvalho et al., 2020). We concluded that grouping projects of the same domain favors the analysis of concerns: (i) we can find more concerns being shared by the projects as they usually have to deal with the same requirements, and (ii) the data tends to be more uniform through time as the shared concerns are usually implemented following the same steps. Table 7 describes the database projects that AKS mined concerns from.

After mining the aforementioned software projects, AKS filled a database with historical information about concerns. Table 8 shows the complete list of concerns. The *Concern* column contains the name of the concerns mined from the database projects. *Purpose* describes each concern's application. The third column, *Average Number of Artifacts*, shows the average number of source code artifacts (.java files of the projects described in Table 7) that contain each concern.

The relationships between concerns and software projects, as seen in Table 8, reveal cases in which not all concerns crosscut, *e.g.*, only one occurrence of "Data Processing" was found in the Heroic database. However, the majority of the associations points to a tendency of concerns originated from third-party components to crosscut, which is shown in the table as cases of concerns that are associated with an average number of artifacts superior to 1, *e.g.*, "Logging" that crosscuts through the analyzed versions of all databases.

We developed an exporter to generate an agreement data set from the information AKS filled in the database. During each round of the study, the raters tagged a data set in the format illustrated in Table 9. The *Project* columns inform the name of the software project. The *Concern* column indicates one particular concern to be analyzed. The list of imports that implement the concern is shown in the *Imports* column. The *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 about 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 ensured that the resulting agreement data set:

- ***Contained a variety of concerns mined from all the aforementioned databases***: the data set should include concerns found in each database. We 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 important 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 data set with samples of source code. This means: at any moment, raters could check the projects' original source code files, from which the concerns were mined.

Table 7. Non-relational Databases (Carvalho et al., 2020)

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

Table 8. Non-relational Databases' Concerns

Concern	Purpose	Average Number of Artifacts(*)					
		J	N	T	H	K	O
Parsing	enables the parsing of source code		1				
Data Processing	enables processing of data set formats, *e.g.*, CSV				1		
Tracing	allows processing of tracing stacks				1		
Directory Management	supports processing of directory structures, *e.g.*, LDAP		4				
Compression	supports data compression				2		
Authentication	enables authentication of users	2					
Cloud Computing	allows communication with cloud services				9.5		
Caching	supports caching strategies				10		
Security	supports security, *e.g.*, authentication		10.2				
Encryption	enables data encryption		1.7				
Validation	supports validation of data					21.8	
Samples/Examples	lessons about how to use APIs		4.5				
Monitoring	enables monitoring of programs' execution				55.6		
RPC Support	adds support to remote procedure calls				5.8		
Mathematical Processing	provides support for complex mathematical calculations	4		13.5			
Geospatial Processing	processes geospatial data	5.3		3			
Dependency Injection	makes injection of components possible				111.2	61.8	
Visualization	enables visualization of data		16		2		
Benchmark	enables benchmark tests	19.2		13.8			
Process Execution	executes external processes, *e.g.*, OS programs				109.7		85.8
Command-line Parsing	automates interpretation of command lines			1	45.2	1	
ElasticSearch Processing	supports processing of document-based information	3.5		3	12.8		
Distributed Computing	adds distributed-computing features	55.2		37			2.7
Web Server Support	allows embedding of web servers		17.2		7.4	1.8	
Graph Computing	supports processing of graphs	168.8		93			1
Text Processing	processes data in form of text	65.8		3	2		
Web App Support	embeds service-client protocols	6.3	105		35	3.3	6
Service-Orientation	enables service-oriented architectures	4	42.8		23	2	
Serialization	supports serialization of data	4.5		7.8	51.2		4
Metrics and Measurement	measures metrical/quality attributes	4		3	151	6.7	
Test	automates self-tests of programs	167.7	1302.5	99.8	80.5	62.8	96.8
Programming Utilities	provides data structures, *e.g.*, lists	119	93	1	37.8	5.5	2.5
Data Format Processing	manages data formats, *e.g.*, xml	5.7	24.7	3	148.3	19.8	24.7
Database	enables communication between clients and databases	597.7	10.5	52.2	10	16.2	80.5
Logging	logs the execution of routines	131.7	9.8	87.8	1.8	41.7	47

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

Table 9. Format of our Agreement Data Set

Project	Concern	Imports	File	DtC	Confirm?	Comment
Titan	Logging	org.apache...	LINK	SLIGHT	YES/NO	...
Neo4j	Test	org.junit...	LINK	SLIGHT	YES/NO	...
Kairos	Database	com.spotify...	LINK	HIGH	YES/NO	...
Heroic	Web App Support	com.google...	LINK	HIGH	YES/NO	...

6.3.2 Data Set 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 *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.

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. In other words, they were not free to fill their own categorization of DtC in the *DtC* column, but 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 data set. We saw this as a way to give them all the time they needed to evaluate the data set with precision.

6.3.3 The Raters

As a way to count on precise technical opinions from software development specialists, we limited the selection of raters to consider that:

- ***The specialists should have experience in the devel-***

opment of software systems: 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 locate and analyze concerns.
- ***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. 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 (Dos Santos and Travassos, 2011).

In total, three specialists participated as raters in the study. One of the raters has been working as a software developer for 14 years and he/she has a specialization in software engineering (*lato sensu*) and is currently enrolled in a master's 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's degree program.

We needed an extra rater during the third round of data set tagging (more details in Section 6.4.3). The rater has a master's degree in computer science and has professional experience too. For the last 8 years, he/she has been participating in software projects as a programmer and software architect/engineer.

All raters have experience in developing Java-based applications. This is important because all projects we analyzed are written in Java.

6.4 Evaluation and Analysis

The goal of **evaluation and analysis** is to describe the data analysis process and its findings (Dos Santos and Travassos, 2011). Accordingly, 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. The next sections bring detailed information about each round of our study.

6.4.1 Round 1

Each rater received an agreement evaluation package which contained: (i) a spreadsheet written in the format exhibited in Table 9; (ii) source code files from which we extracted concerns; and (iii) a copy of one of our papers (Carvalho et al., 2018), to help them to understand some concepts about our concerns mining and analysis strategy.

Results

After both raters sent us the tagged data sets, we ran them through the Cohen's Kappa Agreement Coefficient. We obtained a "fair" agreement between the raters: 0.28 (Strength of Agreement = "Fair"). We then consulted the information they inserted in the *Comment* column of the data set to enhance our method.

Feedback

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 source code in Listing 5[6] shows one of the classes that the raters analyzed.

Listing 5: Annotated Source Code

```
1  package org.kairosdb.core.http;
2  ...
3  import org.junit.After;
4  import org.junit.Test;
5  ...
6  public class WebServerTest
7  {
8   private WebServer server;
9   private Client client;
10
11  @After
12  public void tearDown()
13  {
14   server.stop();
15  }
16
17  @Test(expected = NullPointerException.class)
18  public void
       test_setSSLSettings_nullKeyStorePath_invalid
       ()
19  {
20   server = new WebServer(0, ".");
21   server.setSSLSettings(443, null, "password");
22  }
23  ...
```

Developers added test-related annotations throughout the source code of the **WebServerTest** class: *@After* (line 11) and *@Test* (line 17). They also added import declarations to import *junit* and use its annotations (lines 3 and 4). According to raters, this class' DtC should be categorized as HIGH regarding 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.

As a way to help the raters to analyze source code samples, we took the opportunity to expand AKS' concerns manipulation capabilities. To help raters easily 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. In Listing 6[7], the import of "javax.validation.Valid" is associated with the "Validation" concern (line 4). AKS adds the *@Concern* annotation to indicate the elements of the source code that are used to implement this concern (lines 3 and 11).

[6] The complete source code of Listing 5 can be found here
[7] The complete source code of Listing 6 can be found here

Listing 6: Concern Annotations

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

Arguably, embedding annotations in the source code samples could introduce a bias in our action research study. For example, they 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 identify concerns and their opinions should always prevail above this. In other words, we pointed out 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 linked to some concerns through imported components, but he/she could not determine the value of DtC without the methods' bodies of code. This could compromise the study, as we did not filter *interfaces* out of the agreement data set. We had two options to manage this situation: (i) to perform the evaluation only on artifacts whose classes have methods with a body of code (and exclude *interfaces*); or (ii) to try to persuade the rater to evaluate the DtC of *interfaces*. We opted for the second and confronted him 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 the parameters and the types of 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. At this moment, a relationship is conceptually established between the interface and the concerns.*

6.4.2 Round 2

We carried out the aforementioned improvements and adaptations. We also asked the rater who refused to analyze interfaces to reconsider his 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

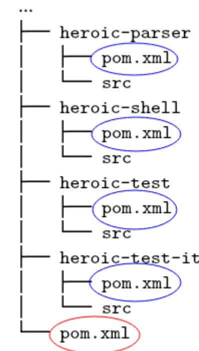

Figure 5. Heroic's POM Files Tree

from those that we sent to them in the first round. Before sending the new evaluation package, we checked it to see if it contained examples/samples related to the problems that the raters identified in the first round. Doing this is important because we wanted them to re-evaluate the same cases and provide new opinions about them.

Results

This time, the processing of Kappa Coefficient gave us 0.26 (Strength of Agreement = "Fair"). So, once again, we counted on the raters' comments to refine our method and AKS.

Feedback

Unfortunately, the rater who did not want to evaluate *interfaces* did not agree with our argument. His 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[8]; (ii) exclude *interfaces* from the agreement data set. The raters would not agree about the categorization of concerns because one of them would never give his opinion about *interfaces*.

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. Some developers scatter the information about third-party components. Figure 5 exemplifies this situation. Developers of Heroic embedded many POM files in its source code. First versions of AKS were 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).

Our method classified the source code in Listing 7[9] as moderately dedicated to implementing the "Database" concern (a moderate DtC), but one of the raters did not agree.

Listing 7: Empty Methods of Classes

```
1  package com.spotify.heroic.statistics.noop;
2  ...
3  public class NoopSuggestBackendReporter
       implements SuggestBackendReporter {
```

[8] The processing or exclusion of *interfaces* can be parameterized during the execution of AKS.

[9] The complete source code of Listing 7 can be found here

```
4     private NoopSuggestBackendReporter() {
5     }
6   ...
```

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** (line 4 to 5). 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.

The raters mentioned cases in which our method failed to associate components with concerns. In Listing 8[10], we illustrate this situation.

Listing 8: Failed Association

```
1 package com.spotify.heroic.http.tracing;
2 ...
3 @Concern(name = "Service-Orientation")
4 import javax.ws.rs.container.
      ContainerRequestFilter;
5 @Concern(name = "Service-Orientation")
6 import javax.ws.rs.container.
      ContainerResponseFilter;
7 ...
8 class OpenCensusApplicationEventListener
      implements ApplicationEventListener {
9     private final Tracer globalTracer = Tracing.
      getTracer();
10    private final OpenCensusFeature.Verbosity
      verbosity;
11 ...
```

The artifact imports several components to implement the "Web App Support" concern, but one of the raters argued that the "javax.ws.rs.*" (lines 4 and 5) should also be associated with this concern. AKS associated all components identified as "javax.ws.rs.*" with the "Service-Orientation" concern only because they are often used to implement restful web services. The rater noticed that the DtC would be different if we also associated the component with the "Web App Support" concern. According to his 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.

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.

We embedded the association between source code artifacts and multiple concerns in the activity 2.2 of our method (Figure 1 of Section 2). Figure 6 illustrates how AKS stores occurrences of overlapped concerns in the database. In the example, AKS associates the source code artifact, "class.java", with two concerns, "Service-Orientation" and "Web App Support" (one artifact to many concerns). Later on, when it is necessary to generate an agreement dataset, AKS encapsulates each concern's specific annotation into the source code.

6.4.3 Round 3

We restarted the study after implementing the aforementioned modifications and additions. We sent a new agreement data set 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.

It is debatable that replacing one of the raters could break the succession of contributions to our study as, with each round, the 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 emphasize that the new rater had enough academic and professional experience to help us concluding our study.

Results

We obtained 0.55 from Kappa (Strength of Agreement = "Moderate") after the raters tagged the data set.

Feedback

Our method applies static analysis on source code artifacts to find concerns and categorize the DtC between them. It is restrained to what we can achieve from the Abstract Syntactic Trees (AST) extracted from the artifacts. This can cause our method to lack the precision that semantically-enriched approaches can accomplish. For instance, we categorized as slightly dedicated the relationship between the source code exhibited in Listing 9[11] and the "Geospatial Processing".

Listing 9: Semantic Processing of Imports

```
1 package org.janusgraph.diskstorage.lucene;
2 ...
3 import org.apache.lucene.index.*;
4 import org.apache.lucene.search.*;
5 import org.apache.lucene.spatial.SpatialStrategy;
6 import org.apache.lucene.spatial.prefix.
      RecursivePrefixTreeStrategy;
7 import org.apache.lucene.spatial.prefix.tree.
      GeohashPrefixTree;
8 import org.apache.lucene.spatial.prefix.tree.
      SpatialPrefixTree;
9 import org.apache.lucene.spatial.query.
      SpatialArgs;
10 import org.apache.lucene.spatial.query.
      SpatialOperation;
11 import org.apache.lucene.store.Directory;
12 import org.apache.lucene.store.FSDirectory;
13 ...
14 public abstract class LuceneExample {
15 ...
```

[10]The complete source code of Listing 8 can be found here

[11]The complete source code of Listing 9 can be found here

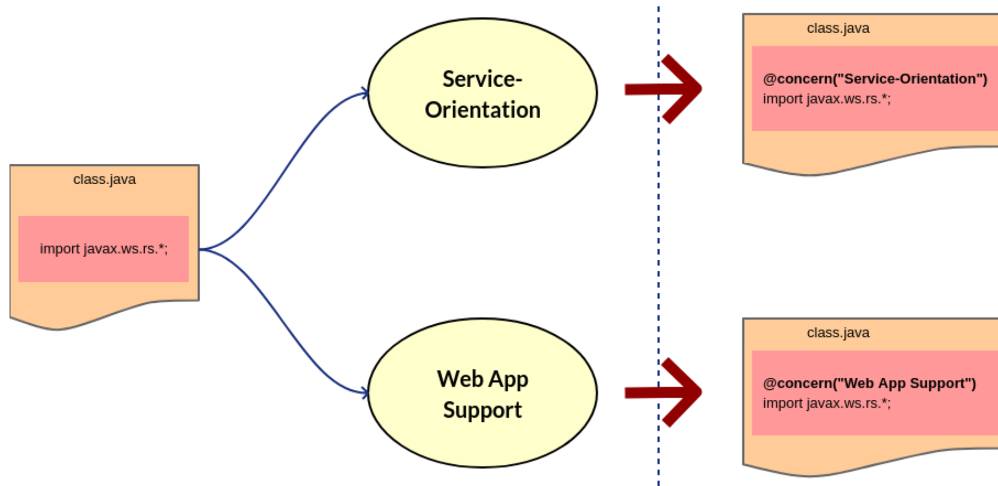

Figure 6. Concerns Overlapping

One of the raters disagreed because he/she took other aspects into account: many occurrences of the word "spatial" (from line 5 to 10) suggest that the artifact focuses on demonstrating how a spatial search is performed. AKS did not process such pieces of information because MVNRepository categorizes "lucene" as a "Full-Text Indexing Library"[12] 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, our method and tool would have an additional resource to refine the categorization of DtC.

AKS determined that the artifact shown in Listing 10[13] is highly dedicated to implementing the "Validation" concern (lines 4, 7, 10, and 17).

Listing 10: Semantic Processing of Annotations

```
1  package org.kairosdb.core.groupby;
2
3  @Concern(name="Validation")
4  import org.apache.bval.constraints.NotEmpty;
5  ...
6  @Concern(name="Validation")
7  import javax.validation.constraints.NotNull;
8  ...
9  @Concern(name="Validation")
10 import static com.google.common.base.
       Preconditions.checkNotNull;
11
12 @GroupByName(name = "tag", description = "Groups
       data points by tag names.")
13 public class TagGroupBy implements GroupBy {
14   @NotNull @NotEmpty private List<String> tags;
15   ...
16   @Concern(name = "Validation")
17   public TagGroupBy(List<String> tagNames) {
18 ...
```

There is a strong relationship between the artifact and "Validation" throughout the 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 12 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 because it points to a purpose other than "Validation". He/she categorized it as MODERATE.

AKS determined that the artifact in Listing 11[14] is highly associated with the "Database" concern.

Listing 11: Semantic Processing of Classes' Names

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

AKS processed the imports in lines 4, 6 and 8 and methods from lines 13 to 15 and 19 to 22 and ended up with a HIGH DtC. One of the raters disagreed and was positive that developers designed the artifact to perform tests. He/She pointed out that the name of the class (in line 10) is more related to "Test" than to the "Database" concern. According to him, DtC should be SLIGHT regarding "Database".

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

[12] Apache describes it as a "full-featured text search engine library written entirely in Java" (https://lucene.apache.org/core/)

[13] The complete source code of Listing 10 can be found here

[14] The complete source code of Listing 11 can be found here

good agreement (Donker et al., 1993; Munoz and Bangdiwala, 1997), 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.

We do not associate obtaining a good strength of agreement to the substitution of one of the raters. In other words, we trust that the new rater fitted our requirements to participate in the study. Additionally, throughout the rounds of interaction, 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.

6.5 A Semi-Structured Interview with the Raters

After finishing the evaluation, we conducted an interview with the raters. We opted for a semi-structured interview 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 had 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 took the opportunity to elucidate some controversial questions that emerged during the rounds of evaluation.

One important result that we discussed with the raters was the fact one of them refused to evaluate the DtC of *interfaces* and methods of *abstract* classes. We must point out that the rater who refused to evaluate the DtC under the mentioned circumstances did not participate in the interview. He/She was the one who stopped responding to our invitations to take part in the third round of evaluation. 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)[15] is a standard for connecting applications to databases. Some of its extensions[16] relies on annotated *interfaces* and developers rarely need to add extra lines of code. According to him, it is important to associate the "Database" concern with JPA-annotated *interfaces*.

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 on many occasions they could not agree with the results listed in the agreement data set 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 can be implemented under different names. Developers 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 how developers refer to concerns.

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, the source code elements that AKS processes to identify concerns (import declarations, attributes/parameters types, variable declarations) tend to be "constant" assets in software making. They are intrinsic parts of many programming languages and programs 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.

6.6 Reflections and Learning

Reflections and learning is responsible for (Dos Santos and Travassos, 2011): (1) exploring the results of the study in comparison to the state-of-the-art, and (2) depicting the learning experience of the participants. It is important to remark that we also learned as researchers, while we gradually became aware of the problems and suggestions reported by the specialists. As stated by Staron (2020), through co-development, researchers and practitioners learn from each other, and thus they develop research results that contribute to both the industrial practice and academic theories, tools, methods, and knowledge development.

Table 10 summarizes raters' opinions and how they impacted our action research study. It took us three rounds of evaluation to reach a good strength of agreement. We achieved a **moderate** agreement (0.55). We believe that our findings can guide other future studies in determining requirements for dealing with the association between concerns and source code artifacts. For instance, fatiguing the raters seems a natural consequence of investigating concerns because the manual identification of concerns in large data sets/databases can be tiresome. Therefore, we consider that developers should be helped whenever they need to perform this kind of activity.

Although our method is practical, it has limitations. For instance, not all concerns which developers implement depend on the use of components. Consequently, our method can grant only a partial view of the collection of concerns that systems encapsulate.

It is important to mention that 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 elements that we discussed in the third round (Section 6.4.3) was already being notified by

[15] https://www.oracle.com/java/technologies/persistence-jsp.html

[16] The rater mentioned the Spring Data JPA (`https://spring.io/projects/spring-data-jpa`)

Table 10. Impact of Raters' Opinions and Reactions

Round/Agreement	Opinion/Reaction	Rationale	Impact
1/0.28 (Fair)	Processing of Annotations	Annotated code is also an indication of associations between concerns and source code	Parsing and Processing of annotations
	Analysis of *interfaces*	*Interfaces* lack methods' body of code. This hinders the identification/verification of concerns and measurement of DtC	Filtering of *interfaces* out of the data set when they are not regarded as valuable for concerns analysis
	Fatigue	As performing the analysis tend to produce a large data set, verification becomes tiresome if done manually	Addition of annotations (@concern) as visual clues to locate concerns and reducing the size of the agreement data set
2/0.26 (Fair)	Processing of multiple POM and Gradle files	Some developers split the information about components through sub-projects and modules	Identification and processing of scattered POM and Gradle files
	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 when they are not regarded as valuable for concerns analysis
	Overlapping of Concerns	Some components can refer to more than just one concern	Enabling the configuration of overlapped concerns
2/0.55 (Moderate)	Processing of source code's semantic elements	Reaching higher strengths of agreement may require combining the processing of AST with semantic elements	Running future studies to advance our method and tool into combining static and semantic analyses

raters since the first rounds (Sections 6.4.1 and 6.4.2). However, during rounds 1 and 2, raters' suggestions about the processing of static aspects of the source code overwhelmed us. As soon as we adapted our study to address such aspects, we began to perceive the semantic facets of concerns mining and analysis.

7 Threats to Validity

Now, we discuss the threats to the validity of our work:

Construct validity: 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 in the information. Although the raters who participated in our action research study (Section 6) have expertise in software development, they did not have any previous contact with the source code samples they analyzed. Imprecision may reside in raters failing to understand how some concerns were implemented.

Internal validity: we determined the thresholds that we embedded in AKS and used to produce our studies' data set. By the time we evaluated our method (in Section 6), we did not take advantage of the raters' expertise to fine-tune our DtC measurement rule. We must conduct new studies to observe how changing the rule's thresholds values impacts our method.

Although reviewing the thresholds of the metrics would be a valuable addition, we do not consider it a major threat. The set of opinions and remarks we obtained from the raters through the rounds of our study (concisely described in Table 10) are still reliable and valuable regardless any potential refinement of the metrics and thresholds. For instance, one of the raters outlined that "Processing of Annotations" is a good source of information about concerns. That would still occur even if we conducted the study under a different metrics/thresholds settings.

External validity: our findings are restrained to the collection of concerns that we extracted from the projects listed in Table 7. We examined real software projects, but we did not cover software domains other than the one that we adopted: non-relational databases. It is desirable to consider a broader context of domains.

Conclusion validity: having two raters per round of evaluation granted us with some advantages: (i) two raters suffice for achieving a high evaluation precision via kappa, and (ii) it helped to reduce our workload as it provided us with a more controlled flow of insights and suggestions. However, we must rely on more specialists to refine our conclusions. In this case, applying Cohen's Kappa to deal with multiple raters (Conger, 1980; Berry and Mielke Jr, 1988) might mitigate such a threat. Alternatively, although it was not a planned move, replacing one of the raters to run our study's third round is one step toward the diversification of our conclusions.

8 Related Work

In this section, we present and analyze related work that proposes techniques related to automatic/semi-automatic identification concerns.

Robillard and Murphy (2002) proposed a way to generate graphs from key abstract structures used to implement concerns. They 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.

Porubän and Nosál (2014) introduced the possibility of projecting multiple concerns over the same pieces of source code. Projections can help developers to perceive which concerns overlap with each other regarding a specific 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.

Juhár 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.

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 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 ReEngineering (AORE) to mine concerns from legacy systems to transform them into Aspect-Oriented applications. AORE requires the identification of concerns before refactoring systems' source code. Specifically, the author's targeted code smells (Fowler and 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. (2017) 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.

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

9 Final Remarks

This paper describes our effort in automating the mining and processing of concerns with the help of third-party components' metadata. We believe that we have been able to circumvent some limitations that we initially identified: (i) the lacking and inadequacies of software documents regarding the representation of concerns, and (ii) error-proneness of approaches that focus on their manual identification. As a result, we created a new method.

Customarily, developers add metadata files to software projects as a way to automate the addition of components. As this has turned into 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 often synchronize the metadata files (POM and Gradle files) with VCS, ensuring that they are updated frequently and evolve together with other source code artifacts and enable the investigation of historical data about how developers integrate components in systems' source code to implement concerns.

To enhance our method, we evaluate it with the help of software development specialists (in Section 6). We meet their opinions regarding 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 systems and concerns.

We believe that our research can support developers to carry important software-related activities, such as: (i) understanding the causes and effects of cross-cutting concerns as their presence can hinder the modularization of systems (Kiczales, 1996; He and Ye, 2015); (ii) refactoring of systems regarding the occurrence of concerns, which has been deemed as an important task in software maintenance (da Silva et al., 2009; Nunez-Varela et al., 2017); 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 et al., 2010).

9.1 Future Work

Future work includes improvements and new evaluations concerning proposed enhancements and the threats to validity that we discussed in Sections 6 and 7. The next paragraphs highlight some of them.

As our method is dependant on the manual association between concerns and third-party components, it is important to reduce the effort required by reaching a consensus about how to name concerns. We did our best to automate the classification of concerns with the help of MVNRepository, as described in Section 2, but our method still lacks proper ways to prevent this task from becoming overly dependent on human assistance.

We empowered our tool with ways to associate source code artifacts with concerns via annotations. We did that with the solo intention of helping the raters to identify the implementation of concerns. However, we did not create an actual concern annotation library/engine to assist developers to define and describe concerns while coding. In other words, currently, AKS can only position the concerns annotations as texts into the source code of software projects. A valuable addition to AKS could include mechanisms to add information about concerns as real annotations.

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, and comments that developers add to the source code.

Although extracting the agreement data sets from the historical data of projects did not affect our action research study (Section 6), we could have relied on a single snapshot of the projects' history. However, we took the opportunity to prepare our method and AKS to calculate DtC through the evolution of systems to support future studies. For instance, we want to see how different development approaches affect DtC through time. How does the DtC of artifacts of projects that follow Test-Driven Development (TDD) principles evolve? Is a HIGH dedication of artifacts to the "Test" concern expected right from the beginning of the development process?

Adding third-party components and metadata about them into systems' codebase is not particular to Java-oriented software projects. It has become a widespread practice and it can enable us to advance our research. In Table 11, we highlight some other programming languages (first column) that usually benefit from metadata files (second column) and online repositories (third column) to embed third-party components in the source code.

Table 11. Metadata and Repositories X Programming Languages

Programming Language	Metadata File	Repository
Javascript	package.json	NPM
Python	requirements.txt	PyPi
PHP	composer.json	Packagist
Ruby	gemfile	Rubygems

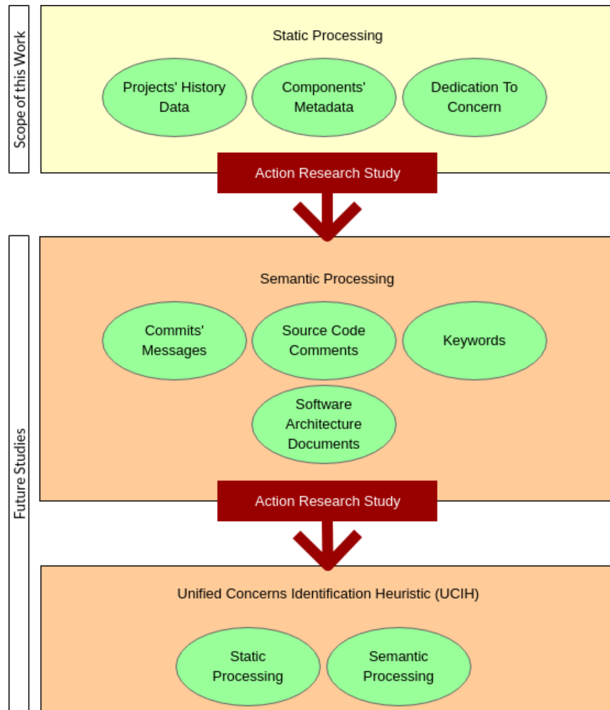

Figure 7. Toward a Generalization

We see potential in all the mentioned examples. It is necessary to generalize our method to deal with other components' metadata files and repositories, as the ones shown in Table 11. Figure 7 illustrates the scope of this work and some future studies that we believe can advance our method and tool. We propose a sequence of studies toward the definition of a Unified Concerns Identification Heuristic (UCIH). We conjecture that we can better reflect the way how developers manage concerns if we merge static and semantic analyses to create a heuristic.

In Section 8, we evaluated some related studies. Some of them mentioned the use of tools to detect concerns and measure their effect on software projects. However, we did not run any of the tools to compare them with AKS. Ideally, we should apply the related tools to extract and analyze concerns from the software projects we based our study on (the ones described in 7). Such a future study must include comparing DtC with the metrics that the related studies' tools are able to extract.

As a result of the interview that we described in Section 6.5, we collected some ideas for future practical applications of our method from raters. 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 perceive that the "Database" concern has a HIGH DtC in the "View" layer. Oppositely, concerns that deal with user interface features (*e.g.*, "UI", "Visualization") are welcomed.

It is not always possible to eliminate unexpected concerns from a particular layer. This comes from developers having to deal with tight schedules and aggressive deadlines, but they may decide to not release a new version of their software if some key concerns appear in the wrong layer. The presence of the "Database" concern in the "View" fits in this category. This can be seen as a security problem, because it may enable hacking via SQL injection. 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.

We believe that analyzing the relationship between concerns and layered software architecture (like MVC) demands incorporating technical debt (TD) concepts into our method. As TD refers to delayed tasks that may require extra effort in the future (de Freitas Farias et al., 2020), releasing systems with misplaced concerns can be seen as developers postponing tasks to correctly place concerns in adequate layers. Figure 8 depicts DtC's potential to spot and calculate TD.

Our method would have to consider one extra parameter to process TD: layers of layered systems. Associating a HIGH DtC to the relationship between the "Database" concern and artifacts of the "View" layer would cause AKS to report a significant case of TD. On the other hand, detecting a HIGH DtC in the artifacts that implement "Database" in the "Model" layer would indicate a LOW (or a nonexistent) TD.

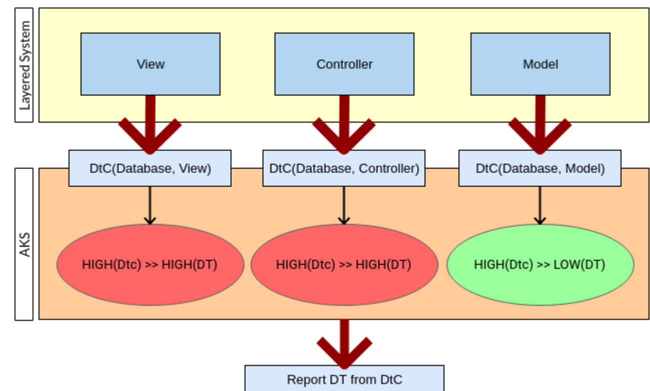

Figure 8. From DtC to Technical Debt

9.2 Replication Package

We encourage the replication and refinement of our studies. So, we have made a replication package available. For AKS' latest updates, visit its public repository. We have made tutorial videos available on the repository to showcase the use of the tool.

References

Abilio, R., Padilha, J., Figueiredo, E., and Costa, H. (2015). Detecting code smells in software product lines – an exploratory study. In *2015 12th International Conference on Information Technology*, pages 433–438.

Abilio, R., Vale, G., Figueiredo, E., and Costa, H. (2016). Metrics for feature-oriented programming. In *2016 IEEE/ACM 7th WETSoM*, pages 36–42.

Adams, B., Jiang, Z. M., and Hassan, A. E. (2010). Identifying crosscutting concerns using historical code changes. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 305–314. ACM.

Agüero, M. and Ballejos, L. (2017). Dependency management in the cloud: An efficient proposal for java. In *2017 XLIII CLEI*, pages 1–9.

Baskerville, R. L. (1999). Investigating information systems with action research. *Communications of the association for information systems*, 2(1):19.

Bellomo, S., Ernst, N., Nord, R. L., and Ozkaya, I. (2014). Evolutionary improvements of cross-cutting concerns: Performance in practice. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 545–548. IEEE.

Bernardi, M. L., Cimitile, M., and Di Lucca, G. (2016). Mining static and dynamic crosscutting concerns: a role-based approach. *Journal of Software: Evolution and Process*, 28(5):306–339.

Berry, K. J. and Mielke Jr, P. W. (1988). A generalization of cohen's kappa agreement measure to interval measurement and multiple raters. *Educational and Psychological Measurement*, 48(4):921–933.

Böhme, R. and Freiling, F. C. (2008). On metrics and measurements. In *Dependability metrics*, pages 7–13.

Brennan, R. L. and Prediger, D. J. (1981). Coefficient kappa: Some uses, misuses, and alternatives. *Educational and psychological measurement*, 41(3):687–699.

Canfora, G. and Cerulo, L. (2005). How crosscutting concerns evolve in jhotdraw. In *13th IEEE STEP'05*, pages 65–73. IEEE.

Carvalho, L. P., Novais, R., and Mendonça, M. (2020). Relationships between design problem agglomerations and concerns having types and domains of software as transverse dimensions. *Under Review for Journal of the Brazilian Computer Society*.

Carvalho, L. P., Novais, R., and Mendonça, M. (2018). Investigating the relationship between code smell agglomerations and architectural concerns: Similarities and dissimilarities from distributed, service-oriented, and mobile systems. In *2018 XII SBCARS*.

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

Conger, A. J. (1980). Integration and generalization of kappas for multiple raters. *Psychological Bulletin*, 88(2):322.

da Silva, B. C., Figueiredo, E., Garcia, A., and Nunes, D. (2009). Refactoring of crosscutting concerns with metaphor-based heuristics. *Electronic Notes in Theoretical Computer Science*, 233:105–125.

Dawson, R. and O'neill, B. (2003). Simple metrics for improving software process performance and capability: a case study. *Software Quality Journal*, 11(3):243–258.

de Freitas Farias, M. A., de Mendonça Neto, M. G., Kalinowski, M., and Spínola, R. O. (2020). Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Information and Software Technology*, 121:106270.

Deacon, J. (2009). Model-view-controller (mvc) architecture. *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf*.

Dias, R. S., de Alcântara dos Santos Neto, P., de Sousa Ibiapina, I. M., Avelino, G. A., and da Costa Castro, O. C. (2019). Effects of visualizing technical debts on a software maintenance project. In *Proc. of the XVIII Brazilian Symposium on Software Quality*, pages 39–48.

Díaz-Pace, J. A., Villavicencio, C., Schiaffino, S., Nicoletti, M., and Vázquez, H. (2016). Producing just enough documentation: An optimization approach applied to the software architecture domain. *Journal on Data Semantics*, pages 37–53.

Dit, B., Revelle, M., Gethers, M., and Poshyvanyk, D. (2013). Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process*, 25(1):53–95.

Donker, D., Hasman, A., and Van Geijn, H. (1993). Interpretation of low kappa values. *International journal of bio-medical computing*, 33(1):55–64.

Dos Santos, P. S. M. and Travassos, G. H. (2011). Action research can swing the balance in experimental software engineering. In *Advances in computers*, volume 83, pages 205–276. Elsevier.

Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N., and Aho, A. V. (2008). Do crosscutting concerns cause defects? *IEEE transactions on Software Engineering*, 34(4):497–515.

Fowler, M. and Beck, K. (1999). *Refactoring: improving the design of existing code*.

Gomes, F., Mendes, T., Spínola, R., Mendonça, M., and Farias, M. (2019). Uma análise da relação entre code smells e dívida técnica auto-admitida. pages 37–44.

Hannemann, J. and Kiczales, G. (2001). Overcoming the prevalent decomposition of legacy code. In *Workshop on Advanced Separation of Concerns*, volume 167.

He, C. and Ye, S. (2015). A method for identification of crosscutting concerns based on goal model and two-state algorithm. In *2015 4th ICCSNT*, volume 1, pages 431–435. IEEE.

Hensler, C. and Stipak, B. (1979). Estimating interval scale values for survey item response categories. *American Journal of Political Science*, 23(3):627–649.

Ibiapina, I., Castro, O., Moura, V., Dias, R., and Neto, P. S. (2018). Tdvision: Um módulo computacional para visualização de dívidas técnicas. In *Anais da IV Escola Regional de Informática do Piauí*, pages 103–108. SBC.

Juhár, J. and Vokorokos, L. (2015). Separation of concerns and concern granularity in source code. In *2015 IEEE 13th International Scientific Conference on Informatics*, pages 139–144. IEEE.

Khomyakov, I., Makhmutov, Z., Mirgalimova, R., and Sillitti, A. (2019). An analysis of automated technical debt measurement. In *International Conference on Enterprise Information Systems*, pages 250–273. Springer.

Kiczales, G. (1996). Aspect-oriented programming. *ACM CSUR*, 28(4es):154–es.

Landis, J. R. and Koch, G. G. (1977). The measurement of observer agreement for categorical data. *biometrics*, pages 159–174.

Longhurst, R. (2003). Semi-structured interviews and focus groups. *Key methods in geography*, 3(2):143–156.

Marçal, I., Garcia, R. E., Eler, D. M., Junior, C. O., and Correia, R. C. (2016). Techniques for the identification of crosscutting concerns: A systematic literature review. pages 569–579.

Mendes, T., Novais, R., Mendonca, M., Carvalho, L., and Gomes, F. (2017). Repositoryminer - uma ferramenta extensivel de mineração de repositorios de software para identificacao automatica de divida tecnica. In *CBSoft 2017 - Sessao de Ferramentas*.

Mendes, T. S., Gomes, F. G., Gonçalves, D. P., Mendonça, M. G., Novais, R. L., and Spínola, R. O. (2019). Visminertd: a tool for automatic identification and interactive monitoring of the evolution of technical debt items. *Journal of the Brazilian Computer Society*, 25(1):2.

Mendes, T. S., Gonçalves, D. P., Gomes, F. G., Novais, R., Spınola, R. O., Mendonça, M., and Salvador, B. (2015). Visminertd: Uma ferramenta para identificação automática e monitoramento interativo de dıvida técnica.

Munoz, S. R. and Bangdiwala, S. I. (1997). Interpretation of kappa and b statistics measures of agreement. *Journal of Applied Statistics*, 24(1):105–112.

Nunez-Varela, A. S., Perez-Gonzalez, H. G., Flores-Puente, Y. T., and Valdes-Souto, F. (2017). Finding core crosscutting concerns from object oriented systems using information retrieval. In *2017 5th CONISOFT*, pages 18–24.

Nuñez-Varela, A. S., Pérez-Gonzalez, H. G., Martínez-Perez, F. E., and Soubervielle-Montalvo, C. (2017). Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197.

Nystrom, M. (2007). *SQL injection defenses*. " O'Reilly Media, Inc.".

Oliveira, P., Valente, M. T., and Lima, F. P. (2014). Extracting relative thresholds for source code metrics. In *2014 Software Evolution Week-IEEE CSMR-WCRE*, pages 254–263. IEEE.

Palyart, M., Murphy, G. C., and Masrani, V. (2017). A study of social interactions in open source component use. *IEEE Transactions on Software Engineering*.

Philippi, C. L. (2021). On measurement scales: Neither ordinal nor interval? *Philosophy of science*, 88(5):929–939.

Porubän, J. and Nosál, M. (2014). Leveraging program comprehension with concern-oriented source code projections. In *3rd Symposium on Languages, Applications and Technologies*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Puhakainen, P. and Siponen, M. (2010). Improving employees' compliance through information systems security training: an action research study. *MIS quarterly*, pages 757–778.

Raemaekers, S., van Deursen, A., and Visser, J. (2017). Semantic versioning and impact of breaking changes in the maven repository. *Journal of Systems and Software*, 129:140–158.

Robillard, M. P., Marcus, A., Treude, C., Bavota, G., Chaparro, O., Ernst, N., Gerosa, M. A., Godfrey, M., Lanza, M., Linares-Vásquez, M., et al. (2017). On-demand developer documentation. In *2017 IEEE ICSME*, pages 479–483. IEEE.

Robillard, M. R. and Murphy, G. C. (2002). Concern graphs: finding and describing concerns using structural program dependencies. In *Proc. of the 24th Internat. Conf. on Software Engineering. ICSE 2002*, pages 406–416.

Rosenhainer, L. (2004). Identifying crosscutting concerns in requirements specifications. In *Proc. of OOPSLA Early Aspects*. Citeseer.

Sant'Anna, C., Figueiredo, E., Garcia, A., and Lucena, C. (2007). On the modularity assessment of software architectures: Do my architectural concerns count? In *AARCH. 07, AOSD*, volume 7.

Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., and Von Staa, A. (2003). On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proc. XVII Brazilian Symposium on Software Engineering*.

Shaikh, M. and Lee, C.-G. (2016). Aspect oriented re-engineering of legacy software using cross-cutting concern characterization and significant code smells detection. *International Journal of Software Engineering and Knowledge Engineering*, 26(03):513–536.

Shatnawi, A., Seriai, A.-D., Sahraoui, H., and Alshara, Z. (2017). Reverse engineering reusable software components from object-oriented apis. *Journal of Systems and Software*, pages 442–460.

Staron, M. (2020). Reporting action research studies. In *Action Research in Software Engineering*, pages 191–213.

Velázquez-Rodríguez, C. and De Roover, C. (2020). Mutama: An automated multi-label tagging approach for software libraries on maven. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 254–258.

Yu, Z., Bai, C., Seinturier, L., and Monperrus, M. (2018). Characterizing the usage and impact of java annotations over 1000+ projects. *arXiv preprint arXiv:1805.01965*.