Carvalho *et al. Journal of the Brazilian Computer Society* (2020) 26:5
https://doi.org/10.1186/s13173-020-00099-y

Journal of the Brazilian Computer Society

RESEARCH Open Access

Relationships between design problem agglomerations and concerns having types and domains of software as transverse dimensions

Luis Paulo da S. Carvalho[1*], Renato L. Novais[2] and Manoel Mendonça[3]

Abstract

Context: Design problems have been recognized as one of the main causes behind the loss of software systems' quality. Agglomerated design problems impact the quality even more. So, organizing and analyzing the relationship between design problems and concerns as agglomerations is a possible way to enhance the identification of defective source code artifacts.

Problem: As different systems evolve in varied manners, it is important to analyze if the evolution of agglomerated design problems can reveal cases of discrepancies and inconstancies through time. We call these cases non-uniformity of agglomerations, and they can prevent the use of agglomerations in approaches to mitigate design problems (e.g., prediction models). To the best of our knowledge, we consider that this problem has not been investigated yet.

Goal: This study aims to comprehend the degree to which the non-uniformity of agglomerations is either the most common or the most exceptional situation during the evolution of software projects. We perform this investigation by grouping software projects under two transverse dimensions: types and domains of software.

Method: To this end, we performed a two-phase investigation: in phase I, we analyzed the historical data obtained from fifteen software projects split as three groups of types of software (distributed, service-oriented, and mobile projects); in phase II, we analyzed the evolution of six projects grouped as two domains of software (graph and timeseries databases). For each phase, we (i) mined instances of a design problem (code complexity) and concerns from the source code of projects, (ii) agglomerated the instances of code complexity around the concerns and analyzed them according to the grouped projects, and (iii) examined the resulting dataset with the help of visualizations and a statistical analysis.

Results/Discussion: Types of software actually shows a tendency to reveal cases of non-uniformity. On the other hand, domains of software show a partial advantage regarding the production of more uniform agglomerations through evolution.

Keywords: Design, Problems, Agglomerations, Concern, Transverse, Dimension

*Correspondence: luispscarvalho@gmail.com,luiscarvalho@ifba.edu.br
[1]Federal Institute of Bahia, Av. Amazonas 3150, Zabelê, 45030-220 Vitória da Conquista, Brazil
Full list of author information is available at the end of the article

Introduction

As software projects evolve, developers usually seek to ensure the quality of the source code. One way to grant the quality relies on finding and managing the occurrence of design problems [1, 2]. A design problem is one kind of phenomenon that affects the maintainability of software projects [3]. For instance, developers can analyze how the presence code smells (a.k.a, smells) increases the complexity of systems [4–6]. According to Fowler and Kent [7], code smells can be defined as a potential indication of problems in the source code of information systems. Thus, it is important to define new approaches to spot them whenever possible. An example of approach can be found in the work of Oizumi et al. [8]: they noticed that when smells interconnect as agglomerations, they show more potential to hinder the quality of software projects. Therefore, developers must investigate how to take advantage from the occurrence of agglomerations in order to improve applications. If single dispersed instances of smells can contribute for increasing the complexity of systems, what can be said when they agglomerate?

Considering this context, we have been investigating the relationship between design problem agglomerations and concerns [9]. In our work, we have adopted the following definition of concern: "anything that stakeholders consider as a conceptual unit" [10, 11]. User interface (UI), exception handling, persistence, test, and logging are examples of concerns. Specifically, we focus on concerns obtained from external components that are made available by third-party developers (more details in the "Extraction of concerns" section).

Additionally, we are interested in analyzing the impact of agglomerated design problems considering three different types of software: distributed, service-oriented, and mobile systems. The rationale behind our research is:

Different types of software may require the implementation of common/shared concerns—for instance, regardless the type of software, developers may always add "Test" routines to validate systems' functionalities.

In opposition, other concerns remain particular to specific types of software—"UI" is not usually implemented by web services, but it is a feature generally found in mobile applications.

If a certain design problem affects concerns that are common to different software systems, it may be advantageous to define global strategies to mitigate such problem—for example, complexity increases in source code artifacts that host code smells. If "Test" is the main concern implemented by many smelly artifacts of systems "M" (a mobile system), "S" (a service-oriented system), and "D" (a distributed system), developers may find it interesting to define a global/general strategy to jointly remove smells from the test-related artifacts of "M", "S", and "D." Developers would have a chance to unify each system's complexity mitigation approaches regardless their distinct types.

In this paper, we extend our use of agglomerations to consider a new aspect: evolution. We investigate if agglomerations are uniformly distributed over software projects' history. Barry et al. [12] and Goulão et al. [13] pointed out that there are differences in the way how systems evolve. Some systems are modified and stay productive for many years, while others are soon replaced or discontinued. Some systems suffer few changes, and others undergo constant changes. As a consequence, it is possible to affirm that software projects do not follow uniform patterns during their lifecycle. This can impact the way how we analyze design problem agglomerations (more details in the "Agglomerations" section). For instance, not all versions of software systems agglomerate instances of design problems. As well, it may not be the case that the agglomerations follow a successive, uniformly spaced pattern through time. If so, the applicability of agglomerations can be undermined, because a lack of uniformity can prevent their use in approaches that require seasonality to explore information patterns, e.g., the creation of prediction models [13].

In summary, our goal is to understand how the lack of uniformity during the evolution of software systems affect the analysis of agglomerations. To this end, we consider an adequate definition of uniformity as follows: it addresses the characteristic of the agglomerations of being non-volatile as a software system evolves through time, enabling the identification of patterns concerning the appearance of design problems. On the other side, non-uniformity refers to situations in which the occurrences of agglomerations are volatile during systems' evolution.

Our study (i) mines concerns and code complexity related to the incidence of code smells from the history of software projects, (ii) agglomerates complexity around concerns, (iii) analyzes combinations between concerns and agglomerations of complexity through the evolution of three types of software project, and (iv) presents a variation of our analysis in which we substitute types for domains of software as a way to group software projects.

We consider both types and domains of software as transverse dimensions. A transverse dimensions is a classification schema which we use to maximize the mining of design problems and concerns. Regarding our study, "types" refer to software systems which are similar to each other regarding targeted platforms (e.g., android apps) and architecture (e.g., distributed and service-oriented systems). We group as "domains" software systems that share the same context of use (e.g., graph-oriented databases). We provide more details about transverse dimensions in the "Types and domains of software as transverse dimensions" section.

Our findings showed that agglomerations follow a non-uniform pattern through the evolution of projects when types of software are adopted as transverse dimension. They also revealed that domains of software are partially better suited to produce cases in which the distribution of agglomerations is more uniform.

The remainder of the paper proceeds as follows: We present a background about our on-going studies in "Theoretical and technical background"; "Study definition" describes the definition of this new study; results are shown in"Results"; in "Discussion," we discuss the implications and effects of the results; "Threats to validity" elicits the threats to validity; "Related work" presents related papers; our final remarks can be found in "Conclusion"; in Abbreviations, we describe some important abbreviations that we mention throughout the paper; Replication and reuse contains instructions about how to replicate our studies and reuse our dataset and tool.

Theoretical and technical background

In this section, we explain an important concept related to our study: the association between design problem agglomerations and concerns. Specifically, we discuss about agglomerations and present our mining and analysis approach.

Agglomerations

Agglomerations is the first concept that we need to describe. Our definition of agglomerations stems from Oizumi et al.'s concept of "semantic agglomerations" [8]: agglomerations are situations in which different source code artifacts affected by design problems address a particular concern. We are interested in the pieces of information systems (e.g., *.java files in java-oriented applications) that contains design problems (e.g., code complexity caused by the presence of code smells) and, at the same time, implement a concern (e.g., "Logging"). Next, we exemplify our use of agglomerations.

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

In [9], we examined the combinations between different types of software and agglomerations. We based our analysis on previous work [1, 8, 14–16] which studied agglomerations of code smells. As a result, we found out that one possible way to visualize and analyze agglomerations is to stratify them as cases of similarities. *Similarities* comprise concerns which are shared by different types of software. Similarities can be divided into *full similarities* and *partial similarities*. Next, we explain and exemplify these types of agglomerations.

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

We need to measure the size of the agglomerations as a way to compare them. We have resorted to the use of *density*. Density is an indirect metric that we calculate in order to compare the agglomerations and to produce visual clues (charts) about how strong/weak is the association between design problems and agglomerations. Figure 3 shows how we use the density of god classes (y-axis) to mine full similarities (x-axis) from systems[1]. In the figure, we used the Lines of Code (LOC) metric to measure the density of the agglomerations. Considering, for instance, the "Test" concern, we found cases of source code artifacts that implement "Test" in all three types of software, and some of these artifacts encapsulate god classes. Specifically, we agglomerated 1185 LOC from god classes of mobile applications, 1099 LOC from god classes of service-oriented projects, and 612 LOC from god classes found in distributed systems (the y-axis shows the total amount of LOC for the entire set of god classes). In other words, the developers of all the three types of software which we examined implemented tests in some source code artifacts, and the first vertical bar in the figure shows the density (or the strength) of the association between the test concern and the code smell that

[1] Table 2 informs the name of the systems, the interval of time during their evolution and the number of files that we mined from them

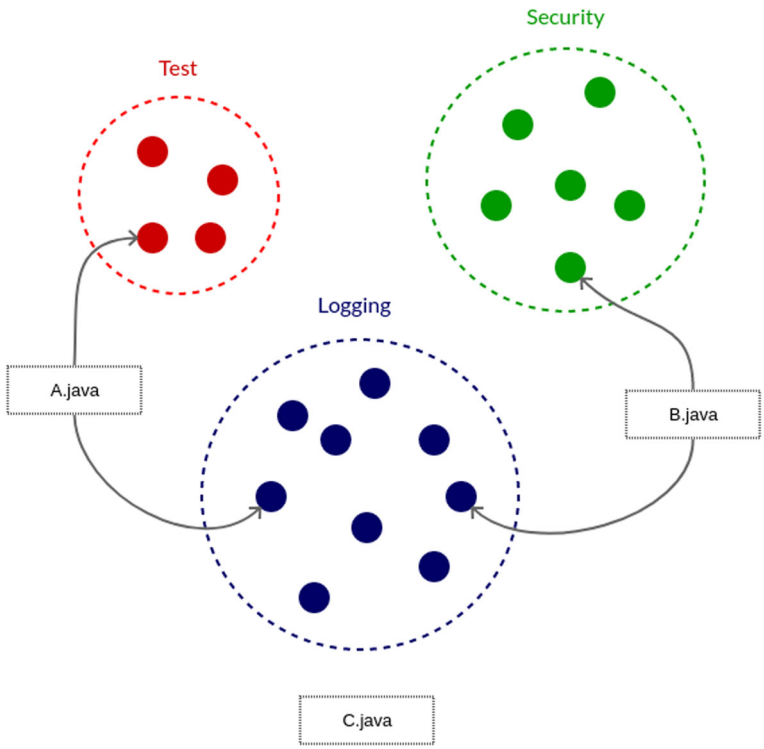

Fig. 1 Agglomerations of project "P"

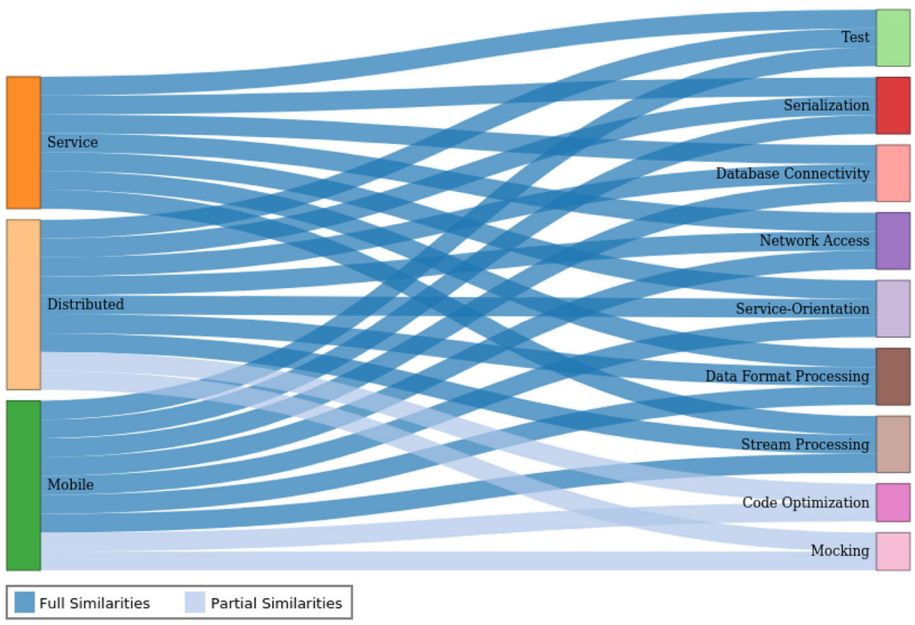

Fig. 2 Types of similarities

these artifacts host: god class. The same happens for other concerns presented in Figs. 3 and 4.

Figure 4 shows partial similarities for distributed and mobile software systems. It has the same agglomerations presented in previous example (Fig. 3) and two new ones which are particular to these two types of software: "Code Optimization" and "Moking." These two concerns are particular to this subset of software projects. This means the service-oriented projects did not contain any artifact that implemented either "Code Optimization" or "Moking" and hosted god classes.

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

Mining and analysis approach

We developed an approach to automate the mining and analysis of the association between design problem agglomerations and concerns [9]. The approach has enabled us to fulfill some demanding characteristics of our studies: (i) we want to mine a large quantity of data to assertively base our results, conclusions, and discussions; (ii) ideally, information about concerns should be extracted from Software Architecture Documents (SAD) or, otherwise, it should be directly provided by the developers of software projects. However, we found neither SADs in the targeted projects, nor we were able to contact the projects' developers. With the purpose of circumventing such limitations, we automated the tasks exhibited in Fig. 5 as much as possible:

Identify concerns—the fulfillment of this task relies on information that developers embed in Project Object Models (POM) and Gradle files. When added to systems, these files are responsible for automating the injection of external third-party components. Mining metadata about components from POM/Gradle files enables us to recover developers' decisions regarding the implementation of concerns (task 1.1). We complement the mining of the metadata by retrieving further information about the components from MVNRepository[2]. MVNRepository is a web portal responsible for indexing useful information about third-party components. Our approach uses MVNRepository to mine categories for components found in projects' POM/Gradle files (task 1.2). As MVNRepository does not provide a category for all components (i.e., many components remain uncategorized), a manual classification of concerns is required (task 1.3). We detail the identification and the classification of concerns from POM/Gradle files in the "Extraction of concerns" section.

Associate concerns with design problems—the main goal of this task is to mine agglomerations of design problems from the source code of systems. At the same time, it must associate the agglomerations with the concerns extracted from the POM/Gradle files. We depend on Repository Miner (RM) to mine metrics and design problems from the historical data obtained from systems' GIT-based repositories [17] (tasks 2.1 and 2.2). After mining the historical data, our approach associates the instances of design problems with the concerns in the form of agglomerations (task 2.3). As the mining of projects' source code produces a large amount of data, RM depends on MongoDb[3] for storage. Mongo's orientation to store data as documents provided us with a flexible and intuitive way to associate information about design problems with concerns.

Export dataset for specialized analysis—our approach is able to externalize the information mined from the source code of projects in a more concise reusable way: as a Comma-Separated Value (CSV) dataset. We believe that exporting the dataset as CSV files maximizes reusability. This type of file can be used by different tools (e.g., spreadsheet editors) to automate analysis in response to varied needs. This task requires the implementation of exporting strategies (task 3.1) to select data from the mined information and generate a specialized dataset (task 3.2), i.e., as the mining of concerns and design problems produces a large database, it is necessary to extract excerpts of data to address systems' specific characteristics and design problems.

Perform specialized analysis—we count on the use of scripts to load (task 4.2) and run the analysis (task 4.3) on the dataset generated by task 3. Externalizing the analysis as scripts favors the expansion of our approach in considering investigations other than the ones contained in this paper. This means other researchers and practitioners can reuse our dataset by writing their own scripts (task 4.1) to perform their own analysis.

We have developed a tool which encapsulates routines to automate the execution of our approach. Our Architectural Knowledge Suite (AKS) can help researchers and practitioners to mine concerns and agglomerations of design problems and to analyze the association between the two.

[2] https://mvnrepository.com/

[3] https://www.mongodb.com/

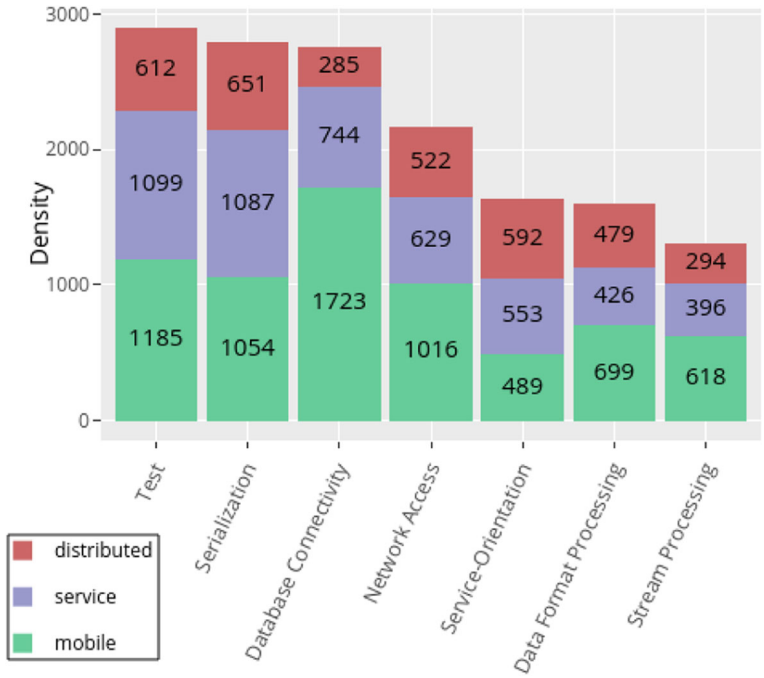

Fig. 3 Similarities between software types and concerns (density by god class) [9]

Extraction of concerns

Development teams often use POM and Gradle files to inject third-party components in software projects [18, 19]. In fact, we were able to find POM and Gradle files in all versions of the projects that we wanted to analyze. Common advantages achieved by reusing third-party components are related to the addition of pre-implemented and pre-tested functionalities. Therefore, they can improve software quality and reduce effort during development [20]. Figure 6 exemplifies how AKS mines concerns from POM and Gradle files: suppose that developers have added components, *org.dbunit* and *org.springframework*, to POM files of two distinct software projects. AKS is capable of processing the POM files to

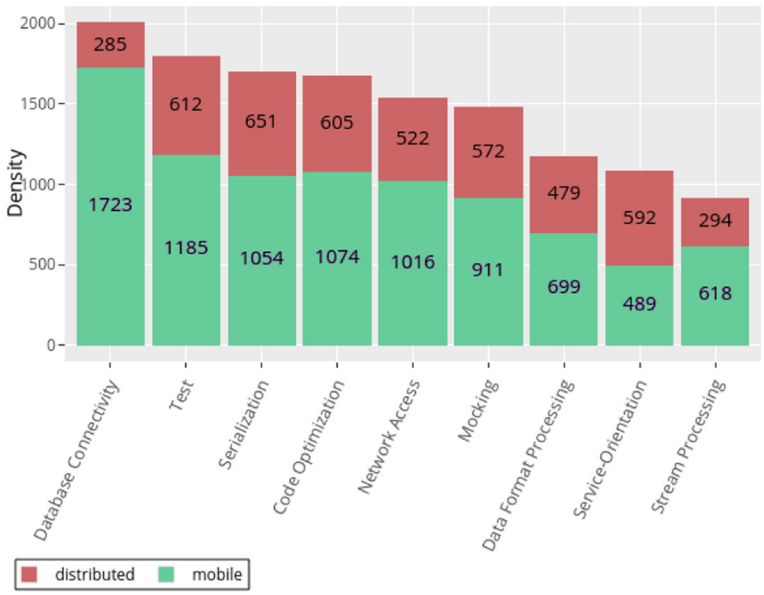

Fig. 4 Partial similarities between distributed and mobile software systems (density by god class) [9]

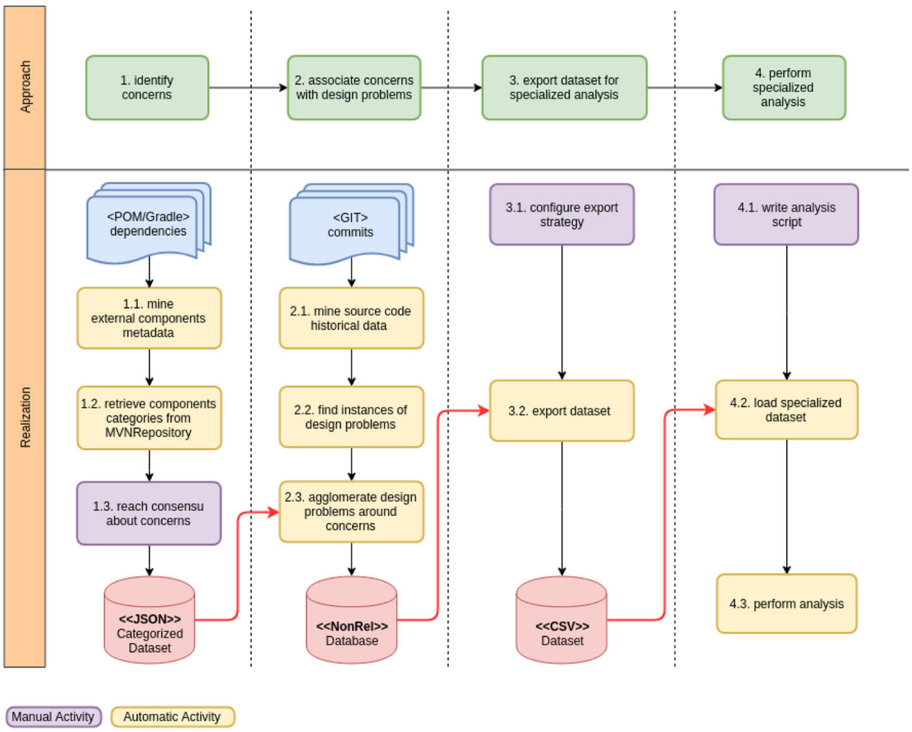

Fig. 5 Mining and analysis approach (adapted from [9])

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 about *org.dbunit* and *org.springframework* from MVNRepository. In Fig. 6, AKS indicates "Testing Frameworks" as the category retrieved from MVNRepository. Later on, we manually fill the most adequate concern which represents each component's category. In the example, we chose "Test".

The manual classification of concerns can be seen as counterproductive and detrimental to the quality of our study. However, we have prepared AKS to minimize the impact of this limitation by parameterizing its execution with the help of configuration files. To assure correctness, we recommend that more than one software development specialist review any dataset generated by AKS. For instance, prior to running this study, we reviewed the manual classification of concerns with the help of two specialists who have both academic and professional experience. They filled the missing categories and concerns after checking each component's web sites, wikis, and other sources of information. Appendix I (on page 29) contains the complete list of concerns mined from the projects that we analyzed.

Study definition

This section details the conducted study. It explains how we measured the density of agglomerations to address a design problem: code complexity. It also presents the research questions that guided this study.

Density of agglomerations from code complexity

We have added routines to AKS to measure the degree to which each concern agglomerates design problems. As a result, AKS is capable of calculating the density of code complexity agglomerations from the Weighted Methods per Class (WMC) metric. In the context of this study, WMC's value is obtained from the sum of the cyclomatic complexity of all methods within smelly classes [4]. We took this decision after analyzing related researches which affirm that code complexity increases significantly in software artifacts that are affected by smells [4–6]. As AKS depends on RM to mine software projects, we are restrained to the set of code smells that it is able to detect: god and brain classes. Considering that RM's god and brain classes detection rules naturally test the values of WMC against thresholds[4], we guarantee that our study's findings are based on relevant cases of code complexity.

[4]More information about RM's code smells detection rules can be found in https://github.com/visminer/repositoryminer/wiki/Available-Code-Smells

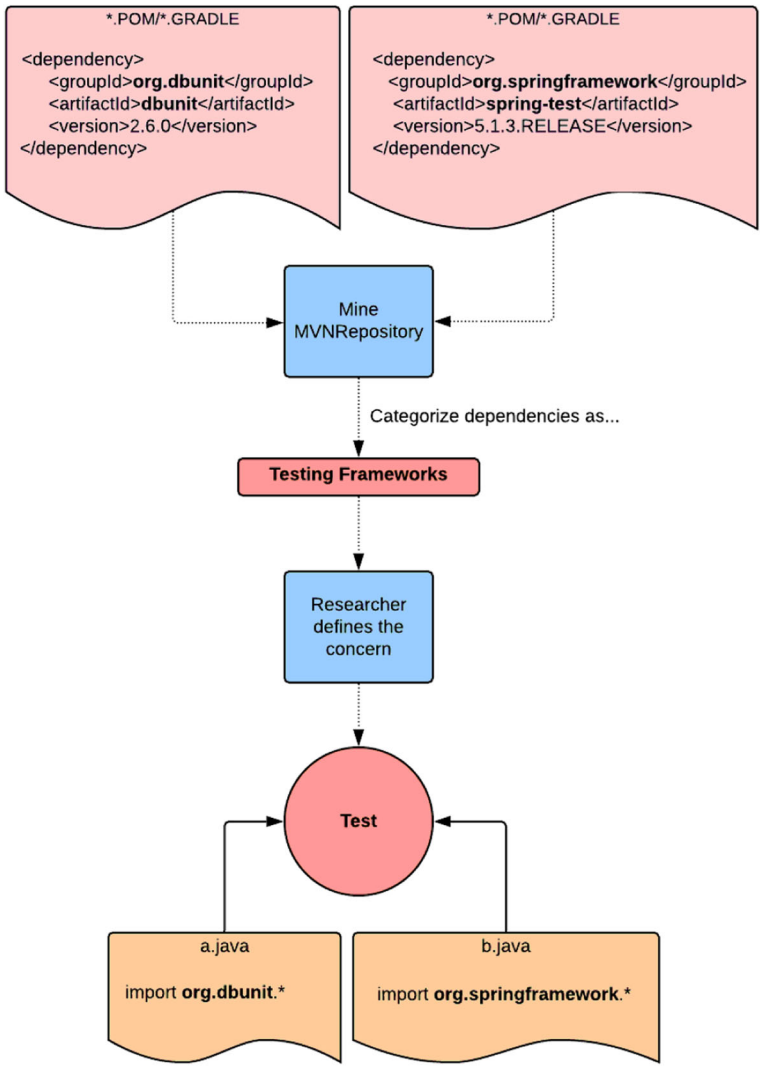

Fig. 6 Mining of concerns (adapted from [9])

The fictitious example in Fig. 7 illustrates how AKS calculates the density of code complexity agglomerations. AKS finds two instances of god classes in "god_class1.java" and "god_class2.java" and one instance of brain class in "brain_class1.java." These are artifacts of project "P." "God_class1.java" is associated with both "Test" and "Serialization" concerns. "God_class2.java" and "brain_class1.java" are associated with the "Serialization" concern only. For each concern, AKS obtains the normalized value of the density by calculating the mean-WMC (m-WMC) of classes. For example, AKS normalizes the density of "Test" by dividing the sum of WMC of all instances of "god_class1.java" by the number of times the smell appeared during the evolution of "P." Then, AKS adds the m-WMC of "god_class1.java" (50) to the density of "Test." Similarly, AKS sums the m-WMC of "god_class2.java" (25), "god_class1.java" (50), and "brain_class1.java" (25) to determine the density of "Serialization" (100). Calculating the normalized density of the agglomerations enabled us to create visualizations to evaluate how design problems agglomerate around concerns. Relying on visualizations to present our findings grants two main advantages:

1 Visualizations are useful to comprehend large amounts of information in a more tangible way [22]. This is the case with the information which AKS mined from the software projects mentioned in this study. By extracting evolutionary data to run our analysis, AKS produced an extensive quantity of data. As a consequence, we applied visualizations to present our results in a more concise and intuitive manner;

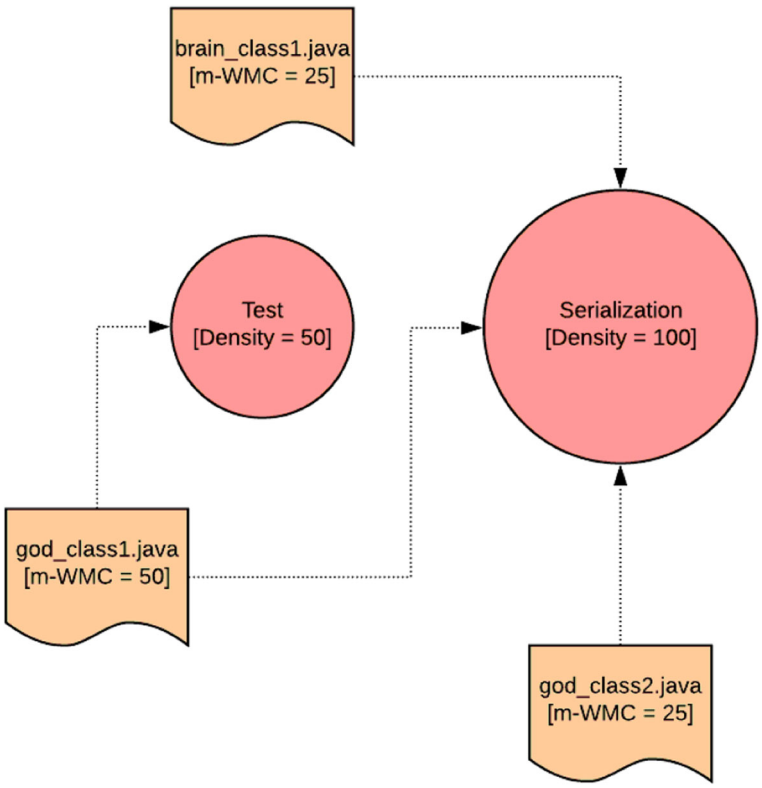

Fig. 7 Complexity-based density of agglomerations of project "P"

2 Considering that spotting design problems in software systems is effort-consuming, such task should be supported by visualizations to inform how problematic source code scatters through projects [23].

Types and domains of software as transverse dimensions

We believe that there is an optimal way to arrange software projects in order to maximize the analysis of concerns. Specifically, we conjecture that it is possible to enhance our investigations after choosing a transverse dimension. We define transverse dimensions as classification schemas which we use to escalate the mining of concerns by joining systems' historical data together. In the context of our research, we cover two transverse dimensions: types and domains of software.

Table 1 explains how we differentiate types and domains of software. We follow a strategy that distinguishes "application domain" from "programming domain" [24]: "application domain" (or domains of software, as we call it) refers to the context of the problem that is addressed by a piece of software, while "programming domain" (or types of software, as we refer to) concerns itself with technical details of implementing applications.

Research questions

We defined two research questions as focus of our investigations:

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

RQ1 is the main question of this study and addresses the problem of non-uniformity of agglomerations. As explained in the "Introduction" section, the norm being the non-uniformity poses a limitation for our approach. It can make its application in the development of methods to

Table 1 Types and domains of software

Dimen.	Definition	Examples
Types	Systems which are similar to each other regarding development platforms and architecture	Android Mobile Apps, Client Server Systems, Services
Domains	Systems that share either the same context of use or fulfill analogous features [21]	E-commerce Applications, Databases, Chatbots

manage design problems difficult. We used the m-WMC metric and the software projects described in Table 2 to answer this question. We applied AKS to analyze a total number of 30921 files through the historical data of the projects.

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

So far, we examined the effects of a particular transverse dimension on the agglomerations of design problems: types of software. However, there is a possibility that other dimensions may present distributions of agglomerations that are more uniform. For instance, domains of software seems to be a fit candidate. We highlight that adopting domains as a transverse dimension to group systems is a new contribution of our studies.

To answer this question, we selected and grouped software projects under a specific domain of software: databases. Databases are good examples of systems in which developers embed components/concerns, and they have been considered a valuable domain for studies [25]. Table 3 describes the two sub-domains of databases that we analyzed: graph and timeseries databases. We used AKS to analyze a total number of 39485 files through the historical data of the database projects.

Table 2 Analyzed projects (transverse dimension: types of software) [9]

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

[a] https://github.com/Netflix/genie
[b] https://github.com/linkedin/pinot
[c] https://github.com/sharding-sphere/sharding-sphere
[d] https://github.com/thinkaurelius/titan
[e] https://github.com/openzipkin/zipkin
[f] https://github.com/opencb/cellbase
[g] https://github.com/nysenate/GeoApi
[h] https://github.com/OHDSI
[i] https://github.com/nysenate/OpenLegislation
[j] https://github.com/openmrs/openmrs-module-webservices.rest
[k] https://github.com/irccloud/android
[l] https://github.com/square/okhttp
[m] https://github.com/nextcloud
[n] https://github.com/square/retrofit
[o] https://github.com/signalapp/Signal-Android

Table 3 Analyzed projects (transverse dimension: domains of software)

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

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

We also ran a statistical analysis on our dataset to find correlative associations between the agglomerations mined from the database projects. The analysis of correlations is another attempt to observe cases of uniformity through the evolution of grouped systems. If the agglomerations of two (or more) software projects correlate around the same concerns, this can be seen as an opportunity for defining common strategies to manage both projects' code complexity. The opposite might indicate that assembling software systems under a transverse dimension is not fully advantageous. In other words, we believe that the use of transverse dimensions is more beneficial if it is possible to perceive a uniform pattern in the way how a design problem affects grouped projects.

Figure 8 summarizes our research. In its first phase, we agglomerate instances of a design problem (code complexity). As a result, we stratify the agglomerations as similarities. Next, we are splitting our dataset into timeseries of systems' releases/versions to analyze how the agglomerations evolve. This means answering *RQ1* has the purpose of knowing if splitting the dataset as a progression of releases/versions reveals that the agglomerations are uniformly distributed through time. If not, we want to test if the adoption of a different transverse dimension (domains of software) tend to produce more uniform agglomerations through time (this being the goal behind *RQ2*).

As last remarks about our study definition, we highlight the following:

1 The software projects that we group as either "types" or "domains," and which we used in our analysis are not mutually exclusive in terms of the concerns they host. This means it was not part of our selection criteria to fully dissociate projects regarding the concerns that developers embedded in them. Consequently, a distributed system may include some concerns which service-oriented systems also have;
2 We evaluated the Titan project as both a type (distributed) and a domain (graph database) of software. So, isolating a system within a particular transverse dimension was not part of our selection strategy either;
3 Figure 2 shows three types of software: service, distributed, and mobile. We spotted the presence of "Service-Orientation" (the concern) in all of them. So, it is arguable that we could have categorized the distributed and mobile systems as services as well, because they externalize some of their functionalities as services. We regard this type of concern as an attempt to add a composition of concerns to software projects. In fact, developers often benefit from libraries that provide many features at once, i.e., function compositions that fulfill several concerns [26]. Including "Service-Orientation" (i.e., a subsystem of services) in distributed and mobile systems is an example of this. In our analysis, we have not differentiated more complex concerns (e.g., "Service-Orientation," "Test") from others that provide simpler functionalities (e.g., "Logging"). We also decided to stick to the way how their developers explicitly classified them according to descriptions that we found in the projects' github repositories (footnotes in Tables 2 and 3).

Results

This section presents the results of our study after using AKS to mine and analyze the projects described in

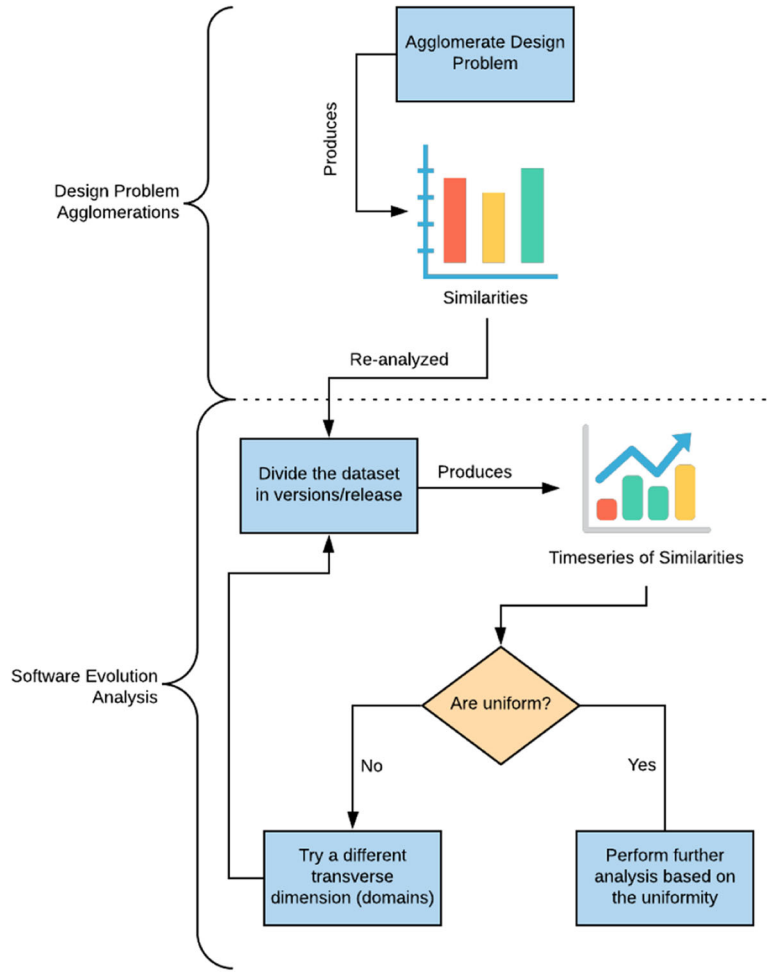

Fig. 8 Summarizing our study definition

Tables 2 and 3. As explained in the "Density of agglomerations from code complexity" section, we rely on visualizations to show the results of our analysis. All visualizations shown in this section (Figs. 10, 11, 12 and 13) follow the same pattern:

1 They show a composition of charts, and each chart displays evolutionary data regarding one software project of one specific type/domain of software;
2 The *x*-axis is divided in versions which AKS mined from the software projects—every single chart shows five versions per project;
3 The *y*-axis quantifies the density of the agglomerations shown in the bar plots—in this case, we used a complexity metric (m-WMC) to calculate the density.

Figure 9 exhibits the evolution of complexity agglomerations considering concerns which are common to three types of software: distributed, service-oriented, and mobile systems. The charts reveal some sources of non-uniformity. Certain projects do not contribute for increasing the density of the agglomerations. For instance, Genie, Signal, and Openmrs do not amount density for any of the concerns: "Data Format Processing," "Database Connectivity," "Network Access," "Service-Orientation," "Serialization," and "Test." Some projects produce agglomerations whose densities are not constant through evolution. ShardingSphere, Titan, IrcCloud, and GeoApi are examples of inconstant projects. We consider that the distributions of agglomerations through the evolution of these project are non-uniform. On the other hand, the evolution of Pinot, Zipkin, Nextcloud, Okhttp, Retrofit, Cellbase, OHDSIweb, and OpenLegislation agglomerate densities throughout their evolution. These are cases of uniform agglomerations.

Figure 10 shows the evolution of agglomerations mined from distributed and service-oriented systems. "Dataset Processing," "Logging," "Messaging," "Process Execution,"

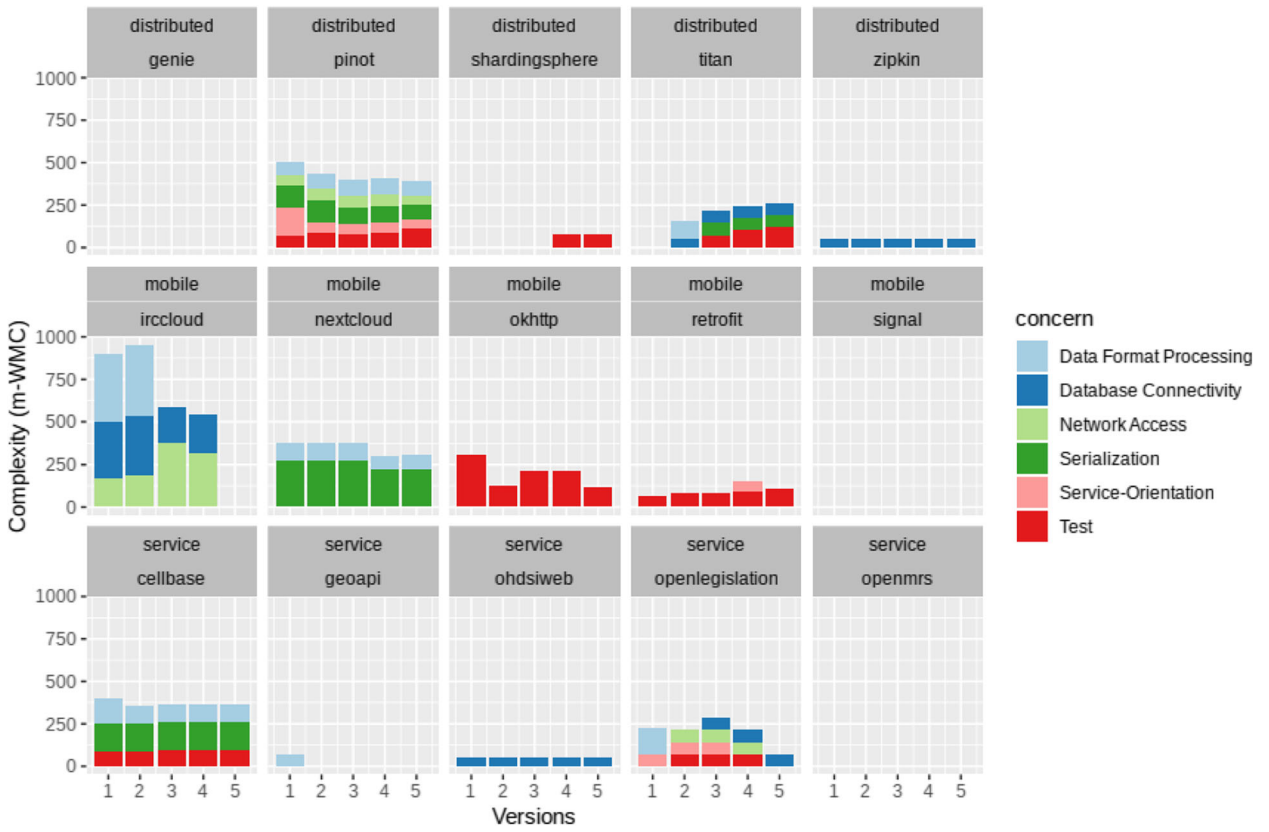

Fig. 9 Uniformity of full similarities (transverse dimension: types of software)

and "Programming Utilities" are concerns which are common to these two types of software. The non-uniformity resides in the fact that some projects (Genie and OpenMRS) do not show any density for these concerns through their evolution. Additionally, the density is not detected in some versions of other projects. This is the case of ShardingSphere, Titan, and Geoapi. Projects like Pinot, Zipkin, Cellbase, OHDSIweb, and OpenLegislation show a more consistent flow of agglomerations as they evolve.

Figure 11 exhibits the partial similarities between mobile and service-oriented software projects. The list of concerns includes the ones exhibited in Fig. 9 and two others that are particular to these two types of software: "Security" and "Stream Processing." OpenMRS shows no agglomeration through its evolution. Agglomerations obtained from IrcCloud, Signal, Geoapi, and OpenLegislation are non-uniform through evolution. NextCloud, OHDSIweb, OkHttp, Retrofit, and Cellbase show more uniform distributions.

Figure 12 shows partial similarities obtained from comparing distributed and mobile applications. They share only one concern beyond those shown in Fig. 9: "Mocking." Genie and Signal do not agglomerate any density through their evolution. ShardingSphere, Titan, and IrcCloud do not show a uniform distribution of agglomerations as they evolve. Pinot, Zipkin, NextCloud, and Retrofit are more uniform.

Now, we present the results regarding the use of domains of software as transverse dimension. Figure 13 shows the collection of full similarities mined from the projects described in Table 3. One noticeable aspect of the figure is that the projects share more common concerns, if compared to our previous analysis of types of software (as seen in Fig. 9). A visual inspection of the figure points out that the agglomerations are more uniformly distributed through the evolution of the systems.

In order to confirm the uniformity, we further examined the data statistically. Considering the possibility of helping developers to manage design problems, it is desirable that the agglomerations are uniformly distributed along the software projects' evolution. For instance, "Test" seems to be uniformly distributed as it affects many versions of the projects, according to a visual inspection of Fig. 13. However, we wonder if the "Test" agglomerations can correlate with each other through a timeseries of versions/releases. As explained, our method can be more advantageous if it

Fig. 10 Uniformity of partial similarities: distributed X service-oriented (transverse dimension: types of software)

enables developers to define strategies to treat the incidence of design problems in a set of projects grouped under the same transverse dimension.

Table 4 contains a series of correlational tests performed on the associations between the agglomerations shown in Fig. 13. For each concern shared by two distinct projects, we tested if the evolution of the agglomerations occurred in concomitance. As the data is not normally distributed, we relied on a non-parametric analysis method (spearman).

The tests exhibited in Table 4 revealed a multitude of results in which high correlations are mixed with low ones. For example, "Test" produced significant correlations between certain pairs of systems (e.g., Janusgraph–Titan, Janusgraph–OpenTSDB), but no significant correlation between others (e.g., Janusgraph–Neo4j, Titan–Timely). As the latter are not exceptional cases, we cannot assume that comparing the evolution of code complexity from any two paired software projects reveals a uniform pattern.

Figures 14 and 15 in Appendix I summarize the relationship between concerns and software projects regarding the two transverse dimensions which we investigated: types (Fig. 14) and domains of software (Fig. 15). In the "Discussion" section, we discuss the impact of our observations and analysis and use them to answer our research questions.

Discussion

We found out that arranging instances of design problems (e.g., code complexity of classes that host code smells) as agglomerations is a possible way to manage problematic source code artifacts [9]. As a result, we ended up with similarities representing the association between design problems and concerns, as the ones exhibited in Fig. 3. However, displaying similarities that way can elude developers in perceiving the data as perfectly seasonal or uniform. For instance, "Test" is shown as a full similarity in the figure. However, considering a deeper analysis, in which different versions of systems are examined, "Test" fails to occur in all versions (as seen in Fig. 9). This means analyzing the evolution of agglomerations reveals cases of inconstancies through time.

Although we are dealing with the same dataset from our previous study [9], this paper presents a new investigation: we are observing the agglomerations through the evolution of software projects. Our findings make us believe that researchers and practitioners should be aware

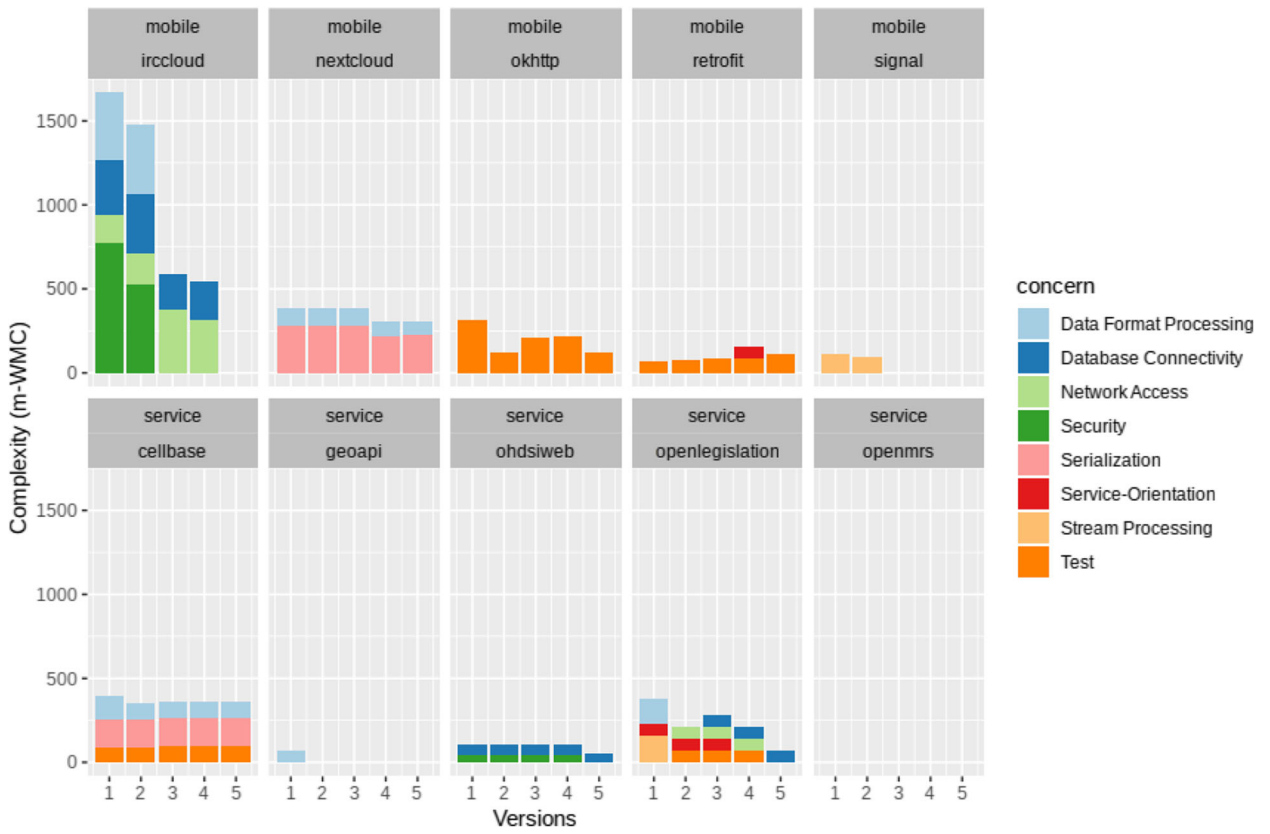

Fig. 11 Uniformity of partial similarities: mobile X service-oriented (transverse dimension: types of software)

about the fact that our approach is sensible to (i) the level of details by which someone presents and analyzes the data—Fig. 3 may hide inconstancies that Fig. 9 reveals, as the latter follows the association between concerns and design problem agglomerations through the evolution of systems; and (ii) the use of a different transverse dimension may promote the uniformity while reducing such inconstancies—Fig. 13 shows timeseries of software projects' releases/versions that are visually more consistent than Fig. 9.

Considering the results that we presented in the "Results" section, we provide the following answers for our research questions:

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

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

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

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

Domains of software like the ones that we analyzed (graph and timeseries databases) seem prone to share more concerns. Most probably, this comes from the fact that a common context of use can lead developers to implement similar functionalities. Eventually, this can even cause the injection of the same (or similar) third-party components. Additionally, domains are also more inclined to produce cases in which

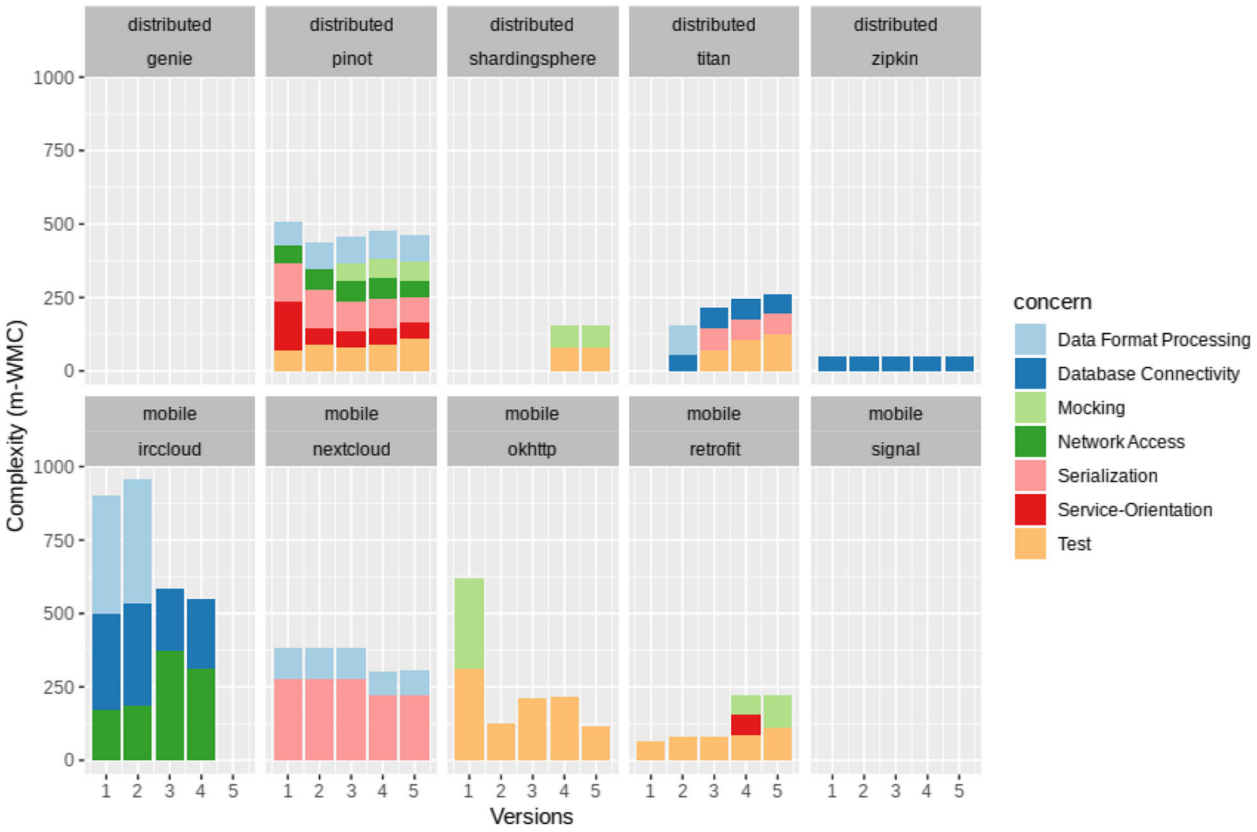

Fig. 12 Uniformity of partial similarities: distributed X mobile (transverse dimension: types of software)

agglomerations are uniformly distributed through the evolution of systems. While this affirmation is assertive considering the visual examination of systems' evolution (shown in Fig. 13), it fails under statistical analysis (Table 4).

We believe that using domains of software to group systems is partially advantageous, and developers may find it useful to spot co-occurrences of design problems that affect different applications. If the co-occurrences are uniform through time, they may define strategies to manage the systems, but they may not count on statistical interpretations of the data to enhance such strategies. For example, agglomerations associated with the "Test" concern were uniformly seen throughout the evolution of four software systems: Janusgraph, Neo4j, OpenTSDB, and Timely. A developer may find it interesting to check if these projects share the same cause for the accumulation of code complexity around "Test". If a common cause is found then he/she can define a generic solution to manage this design problem. However, this advantage is at risk, because the correlation between the evolution of design problems, as they agglomerate around "Test," is not significant considering some paired databases (e.g., Janusgraph X Neo4j).

Lastly, we provide the following answer for *RQ2*: adopting domains of software as transverse dimension to group systems shows a partial advantage in producing cases in which agglomerations are more uniformly distributed. Compared to types of software, grouping projects as domains of software revealed more cases in which concerns are shared. A visual inspection of the agglomerations shown in Fig. 13 revealed that the incidence of code complexity is more uniform across versions of the analyzed systems. As a counterpoint, after the application of correlational tests, the same data failed to produce a consistent statistical uniformity.

We regard domains of software as a better grouping transverse dimension if compared to types of software. Lacking statistical uniformity can be seen as unsatisfactory, but this may not prevent developers from investigating common causes for the agglomeration of design problems around shared concerns. If available, a statistical uniformity (of agglomerations) should be seen as an

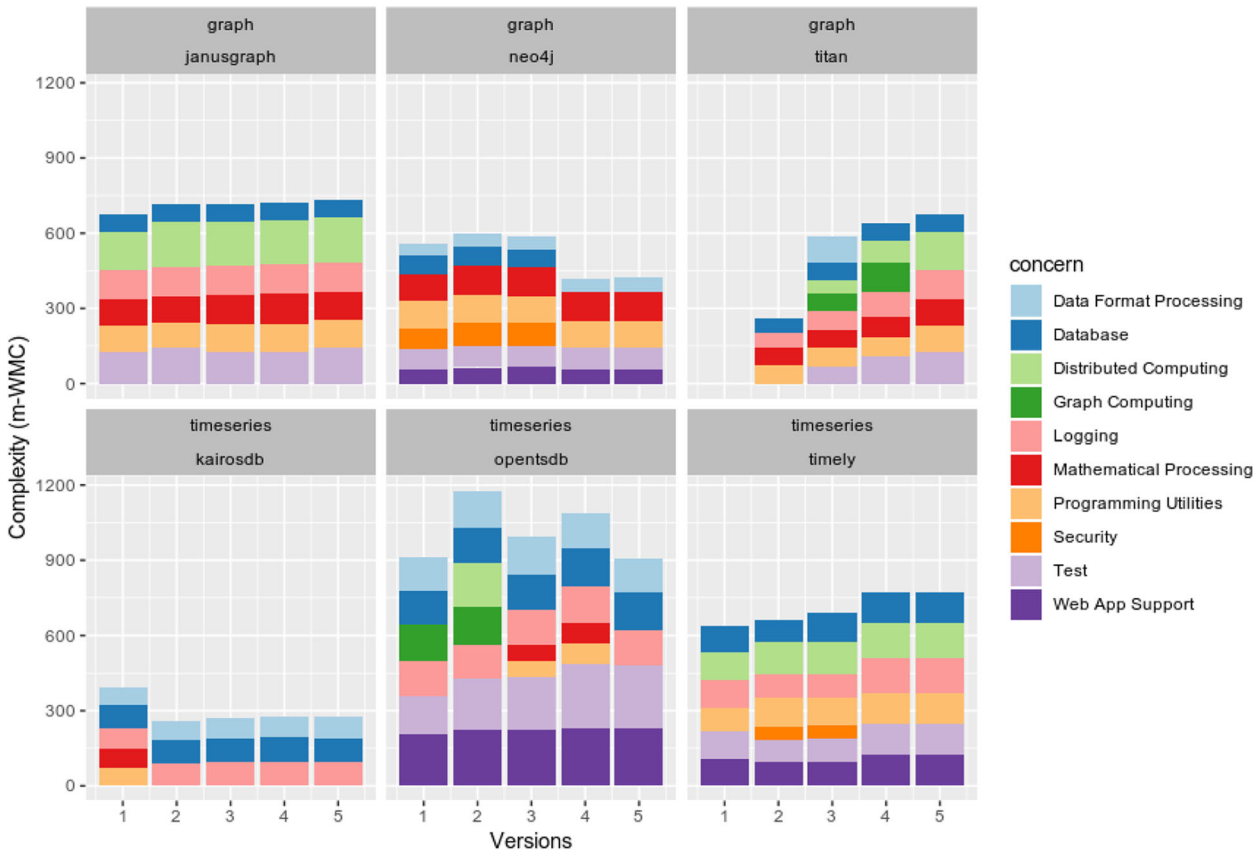

Fig. 13 Uniformity of full similarities (transverse dimension: domains of software)

optimizer for mitigation strategies. If not, our approach is still applicable.

Threats to validity

Now, we discuss the threats to validity we identified in our study:

Construct validity—this type of threat is associated with the relationship between theory and observation. As illustrated in Fig. 6, we mined concerns from third-party components injected in software projects. Later on, our approach used the "import" directive to associate source code artifacts with concerns. However, we cannot guarantee that the imported components are extensively used by the artifacts they are injected in. We have not empowered AKS with features to reject cases in which the imported components are scarcely used to implement a concern. Such features would also have to favor cases in which the bound between components and artifacts spreads over many lines of code or causes a more significant impact. Therefore, using components injection to implement the concept of agglomerations is a potential source of inconsistencies regarding precision in our data. One way to circumvent this limitation requires embedding the mentioned routines in AKS. This will help us to precisely spot situations in which the influence of components (and the association between concerns and agglomerations) is too diluted to be taken in account. Consequently, this might refine our dataset and observations.

Internal validity—internal validity is the extent to which a piece of evidence supports a claim about cause-effect relationships. Even though developers may use a given concern extensively through an artifact, it may be the case that the concern is not alone. Developers may feel like importing several components to support the implementation of many concerns. This imposes a problem: uncertainty regarding the degree to which we can relate the occurrence of code complexity (or any other design problem) to a concern. For instance, source code artifacts are not likely to focus on the use/implementation of the "Logging" concern in isolation. "Logging" tends to play an auxiliary role as developers often implement it as a way to register the activities of other concerns

Table 4 Concern-based correlations between projects

Project	Project	Concern	Corr.
Janusgraph	Neo4j	Database	0.44
	Neo4j	Mathematical Proc.	0.23
	Neo4j	Test	0.28
	Titan	Database	0.45
	Titan	Logging	0.04
	Titan	Mathematical Proc.	0.51
	Titan	Distributed Comp.	0.37
	Titan	Test	0.64
	KairosDb	Database	0.04
	KairosDb	Logging	0.22
	KairosDb	Mathematical Proc.	0.55
	OpentsDb	Database	0.31
	OpentsDb	Logging	0.45
	OpentsDb	Test	0.67
	Timely	Database	0.45
	Timely	Logging	0.56
	Timely	Distributed Comp.	0.09
	Timely	Test	0.70
Neo4j	Titan	Database	0.01
	Titan	Mathematical Proc.	0.18
	Titan	Test	0.80
	KairosDb	Data Format Proc.	0.01
	KairosDb	Database	0.24
	KairosDb	Mathematical Proc.	0.17
	OpentsDb	Data Format Proc.	0.60
	OpentsDb	Database	0.01
	OpentsDb	Test	0.36
	OpentsDb	Web App Support	0.71
	Timely	Database	0.01
	Timely	Test	0.80
	Timely	Web App Support	0.05
Titan	KairosDb	Database	0.25
	KairosDb	Logging	0.08
	KairosDb	Mathematical Proc.	0.17
	OpentsDb	Database	0.00
	OpentsDb	Logging	0.13
	OpentsDb	Test	0.09
	Timely	Database	0.04
	Timely	Logging	0.25
	Timely	Distributed Comp.	0.04
	Timely	Test	0.11
KairosDb	OpenTSDB	Data Format Proc.	0.68
	OpenTSDB	Database	0.22
	OpenTSDB	Logging	0.08
	Timely	Database	0.25
	Timely	Logging	0.25
OpenTSDB	Timely	Database	0.05
	Timely	Logging	0.11
	Timely	Test	0.25
	Timely	Web App Support	0.24

(e.g., logging of steps while accessing a database and the subsequent processing of resultsets). On the other hand, they may specialize other artifacts in performing "Test" routines. This comes from the fact that developers frequently create tests to deal with specific sets of a system's functionalities. Such specialization stems from a good practice related to the automation of tests: single responsibility. This principle is better achieved when tests focus on either very few or on one single behavior of a system [27, 28]. Therefore, it is more likely that a given design problem can be associated with an artifact that implements "Test" (to test one feature/behavior of systems) than with one that hosts "Logging" (in combination with other concerns). We must comprehend the use that developers make of different concerns (in isolation vs in combination) in order to refine our findings.

We have investigated the impact of types of software as transverse dimension [9]. We are now studying the effects of a new possible categorization of agglomerations: domains of software. We claim that our findings can be trusted. However, we still have work to do before we are able to fully testify that a variation from types of software to domains of software grants uniformity. We can achieve this by finding and analyzing other related projects (e.g., other graph and timeseries databases) or adding new subdomains to our investigations (e.g., relational databases, mathematical applications, health care systems).

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

projects' versions to avoid comparing a system's immature versions with another system's more mature ones.

External validity—this threat is related to the degree to which our findings can be generalized. Surely, the number of types and domains of software abounds beyond the ones we examined here. While we believe that our conclusions are assertive, we cannot say that they can embrace other projects or a different set of transverse dimensions. A generalization would require expanding our studies to consider other scenarios. With this purpose in mind, we have been improving our approach, automatizing AKS, and expanding our dataset. This can contribute for advancing and generalizing our studies toward other associations between software projects, design problems, and transverse dimensions.

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

Related work

This section presents related work that have dealt with concerns, agglomerations of design problems, and transverse dimensions.

In a series of papers [8, 14, 15] Oizumi et al. investigated how to use code anomalies to address architectural and design problems in software systems. In their analysis, they grouped software projects based on their types (e.g., web frameworks and middlewares) and design styles (e.g., N-layers and MVC).

Vidal et al. [1] explored the use of agglomerations to prioritize the management of damaging smells. Vidal et al. also emphasize the applicability of architecturally relevant concerns and the use of different design styles (e.g., N-layers and MVC) as transverse dimensions to evaluate the impact of design problems. Both this and the previous work contributed to shape many of the concepts that we have applied in our studies. However, they did not analyze their results considering the impact of transverse dimensions as deeply as we have.

Kazman et al. [29] and Mo et al. [30] interrelated software artifacts to address architectural design flaws through the evolution of analyzed software projects. They pointed out that clustering source code files into special structures called Design Rule Hierarchy (DRS) can help developers to spot parts of systems that are more error and/or change-prone. However, they did not show any results regarding the analysis of concerns and their relationship with transverse dimensions.

Tufano et al. [31] investigated *when* and *why* instances of code smells are introduced in software projects. Their research resembles our own as both seek to understand how the appearance of code smells fluctuates through the evolution of software systems. Tufano et al. provided a broad study about the phenomenon, but they did not focus on grouping the analyzed data around concerns or any specific transverse dimension.

Dosea et al. [16] evaluated the impact of design decisions on the definition of metrics for the analysis of source code. They also relied on concerns (or design roles, as they call it) from different types of software (e.g., eclipse's plugins, android applications, web-based systems) to base their studies.

We also mention our previous study [9] as related work. The study provided us with the first insights about agglomerations and helped us to produce the results discussed in the "Theoretical and technical background" section (Figs. 3 and 4). However, we soon noticed that adopting types of software as transverse dimension could make the applicability of agglomerations less advantageous.

Our work either differs from or expands the aforementioned ones by bringing the association between software evolution and transverse dimensions into focus. We used these two aspects in the analysis of design problem agglomerations. We have also introduced a new mining technique to extract concerns from software projects when their architecture has not been documented.

Conclusion

We introduced some concepts regarding the representation of design problems agglomerations as full and partial similarities [9]. We adopted types of software as a transverse dimension to maximize the number of similarities and pointed out the possibility of using our findings to develop strategies in order to mitigate the bad effects of design problems. Now, we are observing how agglomerations distribute themselves through the evolution of systems as we group them under two transverse dimensions: types and domains of

software. Our analysis provided us with the following conclusions:

1 Software systems of the same domain seem to share more concerns in comparison with systems grouped as types of software. As a consequence, we were able to stratify more agglomerations as similarities;
2 Visual inspections of our charts (Figs. 10, 11, 12 and 13) point out that the agglomerations of design problems are more uniformly distributed through the evolution of software projects grouped under the same domain;
3 After conducting a statistical evaluation (correlational test) on our dataset, we found out that the agglomerations fell short from a desired uniform mathematical precision.

Considering all the aforementioned findings, we believe that using domains of software as transverse dimension can grant a partial but important advantage in comparison to types of software. Domains are advantageous as they are prone to reveal more cases of concerns being shared by software systems. Additionally, the resulting agglomerations of design problems distribute themselves more uniformly through systems' evolution. Although failing to correlate under a statistical analysis can be seen as unfavorable, we do not regard this as so impacting that it may prevent developers from using our approach.

Among our future works, we have the intention to circumvent the problems and limitations elicited in the "Threats to validity" section. As well, we intend to perform other investigations. For instance, we would like to broaden the variability of transverse dimensions. We wonder if other transverse dimensions have potential to reveal more cases of uniformity related to the way how design problems agglomerate around concerns. For instance, development methods might be a good future addition to our studies. How are agglomerations distributed through the evolution of software projects which have adopted Test-Driven Development (TDD) [32] as method? Compared to standardized classic methods, we expect to see TDD-based projects agglomerating more density for test-related concerns right from the beginning of software evolution. Perhaps, projects that follow classical methods (e.g., cascade) [33] might have a tendency to cluster design problems around this concern only during later development phases. If this is the case, design problems which usually plague test artifacts will tend to appear at different moments depending on the chosen method. Developers can use this information to manage test-related design problems when the time is right.

In the "Study definition" section (last paragraph), we highlighted the possibility that embedding third-party components in software projects can lead to composition of concerns [26]. We even mentioned the case of "Service-Orientation" being found in distributed and mobile systems as a way to implement subsystems of services. It seems possible to contrast the effects of using third-party components to inject simpler libraries (e.g., to implement "Logging") from the embedding of more complex ones (e.g., "Geospatial Processing," "Bioinformatics"). We can categorize the later as frameworks and associate them with attempts to add collections of features and concerns that are common to specific domains of software [34]. Thus, as another future work, it is important to evaluate if our approach is adequate to spot developers' strategies to use frameworks in order to compose interrelated concerns.

We believe that we could improve our dataset by making it more reactive to the way how developers contribute for software projects. We are not sure that they usually look around through the evolution of systems to find and manage code smells and the complexity caused by them. Consequently, a pattern in the way how the complex pieces of source code evolve may not be observable. By not reacting to smells, developers may have other motivations to change software projects, and perhaps, the motivations reside in the concerns. Developers of graph databases may tend to focus on improving concerns that are important to this type of software: "Graph Computing," "Mathematical Processing," and "Distributed Computing" (here, we are assuming these ones as examples of most important concerns for the sake of argument). As well, they may consider other concerns as less significant or some concerns will not demand many changes through time. We can raise a new hypothesis then: *To systems grouped under a given transverse dimension there is a subset of concerns which most development effort centrepoints to. Tracking the evolution of such concerns can reveal the actual problems developers deal with in a daily basis.* This may provide a more refined approach to help developers: instead of (firstly) selecting design problems and agglomerating problematic artifacts around concerns, we should better try to find artifacts that implement concerns which attract developers' attention more often (e.g., an exceeding number of commits affecting artifacts that implement "Graph Computing," "Mathematical Processing," and "Distributed Computing" may indicate this) and then identify the design problems that usually affect these artifacts. This can be optimal as it may provide a reactive dataset about problems associated with concerns that developers care the most while discarding/deprioritizing others which are of less impact.

Appendix I – Concerns

Table 5 List of concerns

Concern	**Purpose**	**Found in...**(*)				
		D	S	M	G	T
Service-Orientation	Allows the communication with (web)services or enables a project to provide (web)services	X	X	X	X	X
Data Compression	Supports data compression	X			X	
Configuration	Performs automatic configuration of systems' modules and functionalities	X	X		X	
Metrics and Measurement	Measures systems' metrical/quality attributes	X			X	X
Web App Support	Enables the embedding of server-client protocols	X	X	X	X	X
Graph Computing(**)	Supports the mathematical processing of graphs	X			X	X
Stream Processing	Enables data streaming to-from systems	X		X		
Text Indexing	Processes data in the form of text		X		X	
Geospatial Processing	Supports the processing of geospatial data	X			X	
Test	Automates self-testing of systems' modules	X	X	X	X	X
I/O Processing	Accesses/processes information obtained from I/O devices			X		
Logging	Allows the logging of routines executed by systems	X	X	X	X	X
Data Format Processing	Imports-exports to-from data formats, e.g., xml, json	X	X	X	X	X
Process Execution	Executes external processes, e.g., external programs	X	X	X		X
Report	Enables the preview and printing of reports			X		
Database	Enables the communication with client applications or database servers	X	X	X	X	X
Serialization	Supports the serialization of data	X	X	X	X	X
Benchmark	Enables benchmark tests in applications	X			X	
Distributed Computing	Adds distributed-computing features to software systems	X			X	X
ElasticSearch Processing	Supports the processing of document-based information		X			
Parsing	Makes the parsing and compiling of source code possible	X	X	X		X
Dataset Processing	Enables the processing of dataset formats, e.g., CSV	X	X			
Security	Adds features related to data security, e.g., encryption	X	X	X	X	X
Dependency Injection	Makes the injection of third-party components during runtime possible (hotplug)					X
Caching	Supports the definition of data caching strategies		X	X		
Encryption	Enables data encryption		X	X	X	
Tracing	Allows the processing of tracing stacks			X		
Geometry	Adds functionalities use to process geometric shapes' attributes				X	
Bulding/Deploy	Supports the building and the deployment of systems' releases		X			
Authentication	Enables the authentication of users or client applications		X			
Cloud Computing	Allows the communication with cloud-based applications			X		
Mailing	Enables the sending/receiving of electronic messages		X			
Messaging	Supports the implementation of message queues	X	X			

Table 5 List of concerns *(Continued)*

Validation	Automates validation of data structures during runtime e.g., java beans validation					X
Programming Utilities	Provides special data structures, e.g., list, map, set	X	X	X	X	X
Barcode Reading	Allows the reading of barcodes			X		
Multimedia	Supports the integration with multimedia resources			X		
Bioinformatics	Allows the processing of biological data		X			
UI	Allows the creation of user interfaces	X		X	X	X
Cluster Management	Makes the management of clustered data possible	X				
Mathematical Processing	Provides support for complex mathematical processing	X			X	X

(*) *D* distributed, *S* services, *M* Mobile, *G* graph, *T* timeseries
(**) Not to be confounded with "graphical" computing. A graph is a data structure consisting of vertex and edge sets and a relation that associates each edge with two vertices [35]. Graph computing is the computation of this type of data structure

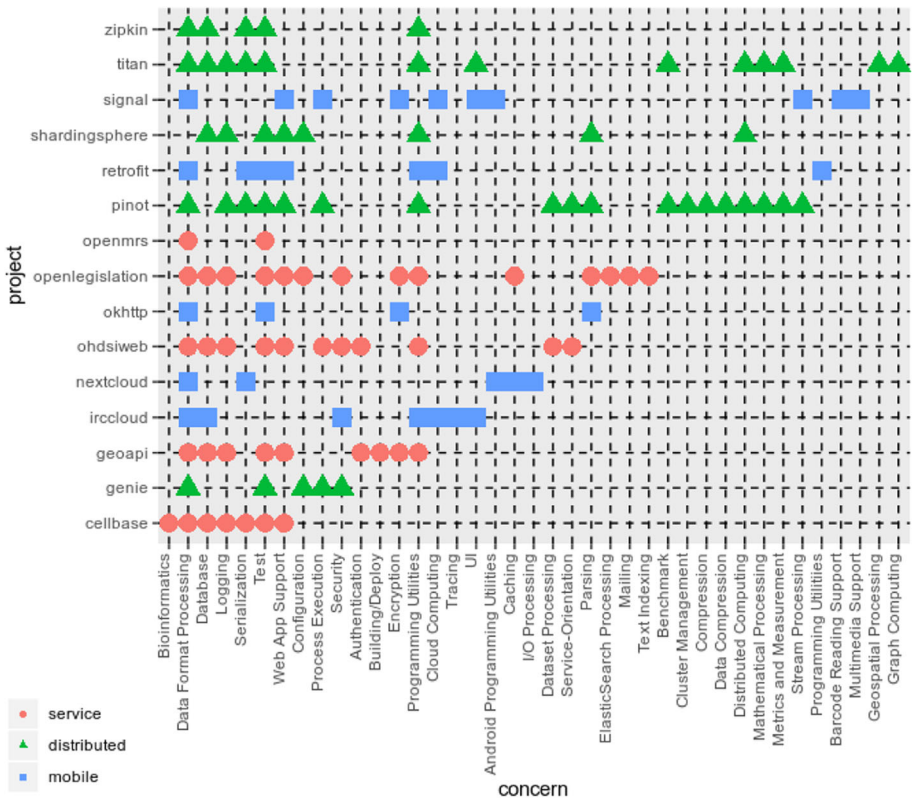

Fig. 14 Concerns per projects (dimension: types)

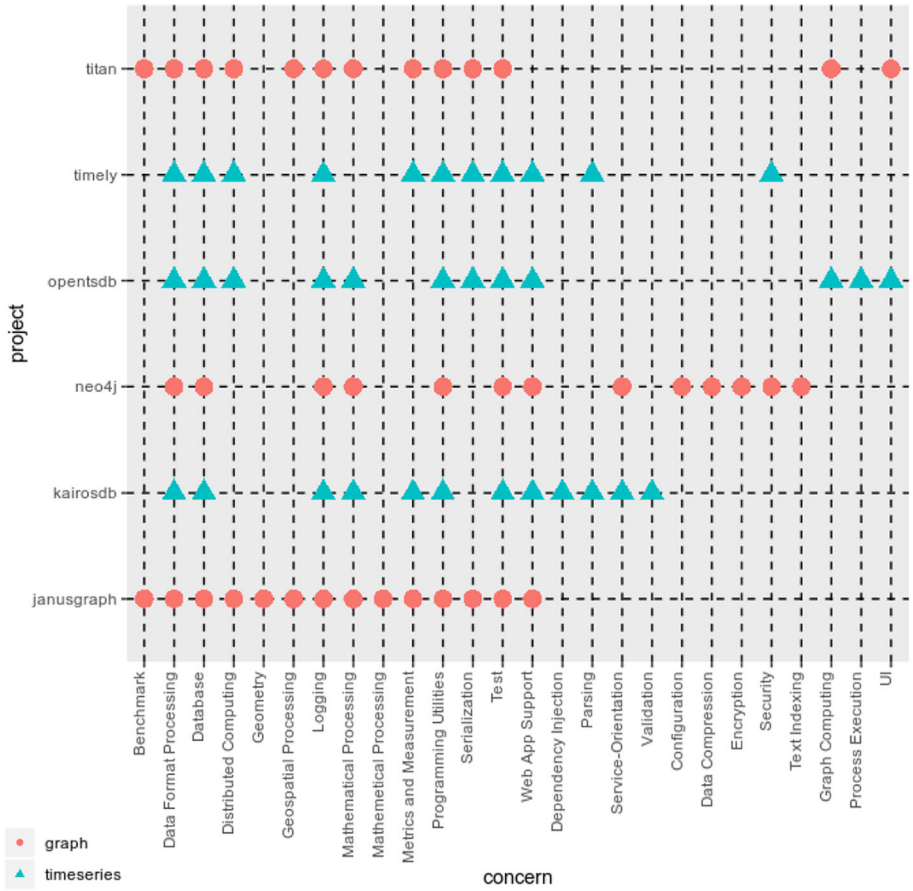

Fig. 15 Concerns per projects (dimension: domains)

Replication and reuse
We have made a replication package available in[5]: www.dropbox.com/s/ss7qwfdse90c73l/replication.zip. The version of AKS used in this study can be found here: www.dropbox.com/s/jk1tzjbtpufg5n0/aks.zip. As our is an on-going research, we have been improving our dataset and AKS. The latest updates can be found in https://gitlab.com/luispscarvalho/AKS. This also implies that we have reviewed our classification of concerns since our previous work [9]. As a consequence, we modified the name of some concerns, e.g., "Database Connectivity" is now called "Database." Additionally, some concerns have been merged into one, as we noticed that they are often used to implement the same set of functionalities. For instance, we grouped "Mocking" and "Test" as "Test," because mocking components are usually applied to add testing capabilities to systems.

Availability of supporting data
Instructions about how to replicate our studies can be found in Replication and reuse.

Abbreviations
Below, we summarize the terms and abbreviations that we used throughout this paper: AKS: Architectural Knowledge Suite. AKS is our mining tool; CSV: Comma-Separated Value. CSV is the type of dataset that AKS generates when it is necessary to analyze design problems with the help of scripts; LOC: Lines of Code. The first metric that we used to calculate the density of agglomerations when we conducted our previous study [9]; POM: Project Object Models. Source code artifacts which we extract information about concerns from; SAD: Software Architecture Document. The first type of artifact which we tried to extract concerns from; TDD: Test-Driven Development. Another transverse dimension that we intend to use to group software projects (future work); WMC: Weighted Methods per Class. WMC is the metric that we applied in this study to measure the density of code complexity agglomerations

Acknowledgements
Not applicable.

Authors' contributions
All authors read and approved the final manuscript.

Funding
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Federal Institute of Bahia, Av. Amazonas 3150, Zabelê, 45030-220 Vitória da Conquista, Brazil. [2]Federal Institute of Bahia, Rua Emídio dos Santos, s/n, 40301-015 Salvador, Brazil. [3]Federal University of Bahia, Av. Ademar de Barros, s/n, 40170-115 Salvador, Brazil.

Received: 15 February 2019 Accepted: 22 June 2020
Published online: 04 July 2020

References
1. Vidal S, Guimaraes E, Oizumi W, Garcia A, Pace AD, Marcos C (2016) Identifying architectural problems through prioritization of code smells. In: 2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS). IEEE, Maringa. pp 41–50
2. Le DM, Carrillo C, Capilla R, Medvidovic N (2016) Relating architectural decay and sustainability of software systems. In: 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA). IEEE, Venice. pp 178–181
3. Trifu A, Marinescu R (2005) Diagnosing design problems in object oriented systems. In: 12th Working Conference on Reverse Engineering (WCRE'05). IEEE, Pittsburgh. pp 10–164
4. Lanza M, Marinescu R (2010) Object-oriented metrics in practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. 1st. Springer Publishing Company, Incorporated
5. Fontana FA, Ferme V, Marino A, Walter B, Martenka P (2013) Investigating th impact of code smells on system's quality: an empirical study on systems of different application domains. In: 2013 IEEE International Conference on Software Maintenance. IEEE, Eindhoven. pp 260–269
6. Velioglu S, Selcuk YE (2017) An automated code smell and anti-pattern detection approach. In: 2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA). IEEE, London. pp 271–275. https://doi.org/10.1109/SERA.2017.7965737
7. Fowler M, Beck K (2018) Refactoring: improving the design of existing code. ISBN/ISSN: 013475770X. Addison-Wesley Professional
8. Oizumi W, Garcia A, da Silva Sousa L, Cafeo B, Zhao Y (2016) Code anomalies flock together: exploring code anomaly agglomerations for locating design problems. In: Proceedings of the 38th International Conference on Engineering. ICSE '16. IEEE, Austin. pp 440–451
9. Carvalho LP, Novais R, 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 Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS). Association for Computing Machinery (ACM), NY. pp 3–12
10. Robillard MR, Murphy GC (2002) Concern graphs: finding and describing concerns using structural program dependencies. In: Proceedings of the 24th International Conference on Software Engineering. ICSE 2002. IEEE, USA. pp 406–416
11. Sant'Anna C, Figueiredo E, Garcia A, Lucena C (2007) On the modularity assessment of software architectures: do my architectural concerns count. In: Proc. International Workshop on Aspects in Architecture Descriptions (AARCH. 07), AOSD vol. 7
12. Barry EJ, Kemerer CF, Slaughter SA (2003) On the uniformity of software evolution patterns. In: Proceedings of the 25th International Conference on Software Engineering. IEEE Computer Society, USA. pp 106–113
13. Goulão M, Fonte N, Wermelinger M, Brito e Abreu F (2012) Software evolution prediction using seasonal time analysis: a comparative study. In: 2012 16th European Conference on Software Maintenance and Reengineering. IEEE, Szeged, Hungary. pp 213–222
14. Oizumi WN, Garcia AF, Colanzi TE, Ferreira M, v. Staa A (2014) When code-anomaly agglomerations represent architectural problems? An exploratory study. In: 2014 Brazilian Symposium on Software Engineering. IEEE, Maceio. pp 91–100
15. Oizumi WN, Garcia AF, Colanzi TE, Ferreira M, Staa AV (2015) On the relationship of code-anomaly agglomerations and architectural problems. J Softw Eng Res Dev 3(1):11
16. Dósea M, Sant'Anna C, da Silva BC (2018) How do design decisions affect the distribution of software metrics? In: 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC). IEEE, Gothenburg, Sweden. pp 74–7411
17. Mendes T, Novais R, Mendonca M, Carvalho L, 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
18. Agüero M, Ballejos L (2017) Dependency management in the cloud: an efficient proposal for java. In: 2017 XLIII Latin American Computer Conference (CLEI). IEEE, Cordoba. pp 1–9
19. Palyart M, Murphy GC, Masrani V (2018) A study of social interactions in open source component use. In: EEE Transactions on Software Engineering. Vol. 44. No. 12. IEEE. pp 1132–1145
20. Shatnawi A, Seriai A-D, Sahraoui H, Alshara Z (2017) Reverse engineering reusable software components from object-oriented apis. J Syst Softw 131:442–460
21. Gomaa H, Farrukh G (1997) A software engineering environment for configuring distributed applications from reusable software architectures. In: Proceedings Eighth IEEE International Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering. IEEE, London. pp 312–325
22. Francese R, Risi M, Scanniello G (2015) Enhancing software visualization with information retrieval. In: 2015 19th International Conference on Information Visualisation. IEEE, Barcelona. pp 189–194
23. Oizumi W, Sousa L, Garcia A, Oliveira R, Oliveira A, Agbachi O, Lucena C (2017) Revealing design problems in stinky code: a mixed-method study.

[5]It contains a README file with instructions about how to (re)use the replication package

In: Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse. p 5
24. O'Brien MP, Buckley J, Shaft TM (2004) Expectation-based, inference-based, and bottom-up software comprehension. J Softw Maint Evol Res Pract 16(6):427–447. Wiley Online Library
25. Tempero E, Anslow C, Dietrich J, Han T, Li J, Lumpe M, Melton H, Noble J (2010) The qualitas corpus: a curated collection of java code for empirical studies. In: 2010 Asia Pacific Software Engineering Conference. IEEE, Sydney. pp 336–345
26. Busch A, Fuchß D, Eckert M, Koziolek A (2019) Assessing the quality impact of features in component-based software architectures. In: European Conference on Software Architecture. Springer, Cham. pp 211–219
27. Runeson P (2006) A survey of unit testing practices. IEEE Software 23(4):22–29
28. Bowes D, Hall T, Petric J, Shippey T, Turhan B (2017) How good are my tests? In: 2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM). IEEE, Buenos Aires. pp 9–14
29. Kazman R, Cai Y, Mo R, Feng Q, Xiao L, Haziyev S, Fedak V, Shapochka A (2015) A case study in locating the architectural roots of technical debt. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE, Florence Vol. 2. pp 179–188
30. Mo R, Cai Y, Kazman R, Xiao L (2015) Hotspot patterns: the formal definition and automatic detection of architecture smells. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture. IEEE, Montreal. pp 51–60
31. Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2015) When and why your code starts to smell bad. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE, Florence. pp 403–414
32. Bajaj K, Patel H, Patel J (2015) Evolutionary software development using test driven approach. In: Computing and Communication (IEMCON), 2015 International Conference and Workshop On. IEEE, Vancouver. pp 1–6
33. Sommerville I (2010) Software Engineering. 9th edn.. Addison-Wesley Publishing Company, USA
34. Vale T, Crnkovic I, De Almeida ES, Neto PADMS, Cavalcanti YC, de Lemos Meira SR (2016) Twenty-eight years of component-based software engineering. J Syst Softw 111:128–148
35. B. West D (1996) Introduction to graph theory, vol. 2. Prentice hall, Upper Saddle River, NJ

Publisher's Note