Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A study on correlations between architectural smells and design patterns[☆]

Ilaria Pigazzini[a,*], Francesca Arcelli Fontana[a], Bartosz Walter[b]

[a] *University of Milano - Bicocca, Milan, Italy*
[b] *Poznan University of Technology, Poznan, Poland*

A R T I C L E I N F O

Article history:
Received 27 November 2020
Received in revised form 3 March 2021
Accepted 15 April 2021
Available online 19 April 2021

Keywords:
Architectural smells
Design patterns
Architectural smells and design patterns relationships

A B S T R A C T

Design patterns are recommended solutions for typical software design problems, with an extensively studied and documented impact on various quality factors. Flaws in design at a higher levels of abstraction are manifested in architectural smells. Some of those smells, similarly to code smells, can reduce the expected advantages of design patterns or even prevent their proper implementation. In this paper we study if and how design patterns and architectural smells are related, and how this knowledge could be exploited in practice. We present an empirical study with an analysis of 16 design patterns and 3 architectural smells in 60 open source Java systems. We analyze their diffuseness and correlation, and we extract association rules that describe their presence and dependencies. We demonstrate that there exist relationships between architectural smells and design patterns, both at the class and package levels. Some smells appear falsely positive, as they result from conscious decisions made by programmers, while the application of some patterns can be a cause of certain smells. Our results provide evidence that design patterns and architectural smells are related and affect each other. With knowledge about the relationships, programmers can avoid the side effects of applying some design patterns.

1. Introduction

Design patterns (DPs) (Gamma et al., 1995) are generic, reusable solutions for recurring software design problems. Their adoption is widely recommended, since they capture verified, distilled knowledge based on experience. They also explicitly identify the trade-offs between their advantages and shortcomings, which help developers in making informed, conscious decisions concerning software design. Architectural smells (ASs) (Garcia et al., 2009) are surface manifestations of strategic decisions concerning software architecture which negatively impact the internal quality of a software system. As such, ASs can be considered as a counterpart to code smells (CSs) (Fowler, 1999) at the architectural level. They are symptoms of issues at the design and architectural levels, which likely lead to software architecture erosion and an increase of technical debt (Nord et al., 2012).

Those concepts seem not only unrelated, but even also disjoint, as they represent two fundamentally different approaches to software quality. DPs provide *recommendations* for software design issues, securing some additional quality properties, like flexibility, reusability or extensibility. They are directly applicable, as they include instructions for implementation. On the other hand, ASs represent *warnings* that indicate the possible presence of deeper quality issues that cannot be directly identified or whose identification is hindered. Additionally, DPs and ASs refer to different levels of abstraction: while DPs address the *tactical* level, solving problems with a limited scope of methods and classes, ASs usually comprise more comprehensive issues involving modules, packages or components, having a more *strategic* impact on the entire system. As a result, even if ASs and DPs may be collocated or related at the structural level, there is currently no evidence that their interactions have an impact on the quality.

However, a more thorough examination reveals some scenarios in which the *relationships* between DPs and ASs may affect the system in meaningful ways, reaching far beyond their individual impact. In this context, by “relationship” we mean a co-occurrence of DPs and ASs within the same software dependency connecting two architectural components (e.g., Java classes and packages).

First, the use of a DP is a deliberate decision of a programmer to apply a specific solution to a given problem. Each pattern has predictable consequences, showing the benefits and trade-offs of its application. However, they can be reduced, changed or

☆ Editor: Shane McIntosh.
* Corresponding author.
E-mail addresses: i.pigazzini@campus.unimib.it (I. Pigazzini), arcelli@disco.unimib.it (F.A. Fontana), bartosz.walter@cs.put.poznan.pl (B. Walter).

even reversed due to interfering factors that change the structure or behavior of the pattern. For example, the advantages of applying the Template Method design pattern to structure inheritance hierarchies are diminished if the respective methods overriden in the subclasses do not follow the Liskov Substitution Principle (Liskov and Wing, 1994) or are affected by a Tradition Breaker code smell (Lanza and Marinescu, 2006). Similarly, an unidirectional dependency structure in the Observer pattern could be broken by introducing cycles to it (Fontana et al., 2016). CSs have already been found to interact with patterns (Walter and Alkhaeir, 2016), thereby affecting their prevalence. Therefore, we may conjecture that various ASs, frequently referring back to an architectural planning phase, can also impact, restrict or even prevent the application of specific DPs.

Secondly, the relationship between ASs and DPs can also be reverted: smells can be manifestations of defects in pattern instances (Moha et al., 2005), e.g., a flawed Chain of Responsibility DP with two-directional dependencies will result in the Cyclic Dependency AS. In that case, effort invested in removing the smell can also fix the pattern implementation.

Smells are manifestations of deeper design or architectural issues, but this relationship is inevitably affected by uncertainty: not every smell refers back to a real underlying problem. As a result, in some cases they can be conscious and accepted effects of a DP, which would make them false positives. This effect has been already studied with regard to code smells: the presence of a smell can be attributed to the application of a DP (Fontana et al., 2016). At an architectural level, Cyclic Dependencies smell (Arcelli Fontana et al., 2016), describing circular references among components, can be an effect of implementing callbacks (Suryanarayana et al., 2014), a common notification mechanism in GUI-related applications. Also a Hub-Like Dependencies AS, describing architectural components with numerous dependencies, can be incorrectly detected, when actually being the implementation of Controller or Orchestrator patterns (Fowler, 2002).

Following from these observations, we can conclude that the effects of the mutual interactions between an AS and a DP may be non-trivial and multi-aspect, and we can expect their significant impact on selected quality characteristics. For that reason, the study of these relationships deserves a closer analysis to determine how and to what extent these concepts affect each other. Although a link between code smells and DPs has been already analyzed (Walter and Alkhaeir, 2016; Jaafar et al., 2013a; Sousa et al., 2019), no empirical study on ASs has been presented. The aim of our study is to investigate this subject.

Since several ASs concern dependencies between architectural components, and DPs strictly influence the design of such dependencies, we can hypothesize that the eventual relationship between the two phenomena will be revealed by analyzing these dependencies.

For this reason we provide and use a dataset of 60 open source Java projects which reports classes and packages having ASs and DPs, together with the dependency information. The dataset has been created with two existing academic software analysis tools: *Pattern4* (Tsantalis et al., 2006) for DP detection and *Arcan* (Arcelli Fontana et al., 2017) for AS detection.

The results of our study can draw developers' attention to the key parts of software systems, and can enable them to incorporate knowledge of the extracted AS-DP relationships.

The main contributions of this paper are three-fold:

- we provide *a dependency dataset*, with information regarding object-oriented dependencies, ASs and DPs;
- we present *a study on the frequency and correlation of ASs and DPs in 60 open source projects*, based on statistical analysis, e.g., correlation analysis and mining association rules;
- we formulate *useful hints for developers and researchers*, to help them avoiding potentially hazardous combinations of DPs and ASs, as well as to enhance the detection strategies for ASs.

The outline of this paper is the following: Section 2 presents related work concerning empirical studies on DPs and ASs; Section 3 describes the empirical study design; Section 4 shows the results of the analysis; Section 5 presents a discussion on the collected data with respect to the research questions; Section 6 reports the threats to the validity of our work, and finally Section 7 presents our conclusions and suggests some possibilities for further research.

2. Related work

The direct relationship between ASs and DPs has not been widely discussed in the literature as yet. For that reason, in the subsequent sections we present an overview of empirical studies regarding these issues separately.

2.1. Empirical studies on interactions between code smells and design patterns

A few works investigated the role of Code Smells (CSs) and their interactions with DPs. Walter and Alkhaeir (2016), based on an empirical study of seven CSs and nine DPs identified in two long-evolving, mid-sized Java applications (Apache Maven and JFreeChart), observed that classes participating in DPs are affected by CSs less frequently than other classes. In a similar work, Cardoso and Figueiredo (2015) discovered co-occurrences of CSs and DPs in same classes. They also identified cases of DP misuse or overuse: in those cases, DPs were found to have a negative impact on the quality. This thread was further explored by Sousa et al. (2019), who looked for co-occurrences of DPs and CSs to identify factors that contribute to such correlations. They found that the application of a DP does not necessarily prevent the presence of CSs; in particular, Adaptor-Command, Proxy, and State-Strategy patterns were highly correlated with smells. As root causes for such cases they indicated poor planning and inadequate application of the DP.

A slightly different topic was examined by Jaafar et al. (2013b), who presented a study of three different open source systems, focused on an analysis of the CS dependencies. They showed that classes that have dependencies with CSs are usually more fault-prone than others. They also found that classes involved in static relationships between CSs and DPs are more change-prone, but less defect-prone than other classes affected by CS.

2.2. Empirical studies on design patterns

The impact of a DP on selected quality characteristics has been the subject of several studies. For instance, Aversano et al. (2009) analyzed the frequency and scope of changes in code with DPs in three open source projects. They observed that patterns closely related to the application's purpose (specifically, Composite, Prototype, and Adaptor-Command) are modified more frequently than others. This shows that DPs affect maintainability in terms of change-proneness, an aspect which is also studied for ASs (Mo et al., 2015): if certain files change more frequently than others, it may be a sign of maintainability issues.

The link between DPs and defects was studied by Vokac (2004). He observed that the use of patterns does not always result in fewer defects; specifically, Singleton and Observer appeared more defect-prone, whereas Factory-related patterns had lower defect numbers.

A more recent work from Alkhaeir and Walter (2021) found that classes participating in a design pattern and also affected by code smells attract more defects than non-smelly pattern classes. They conclude that the presence of code smells in design pattern classes appears to be a contextual factor affecting the defect-proneness of the subject code.

Izurieta and Bieman (2013) and Feitosa et al. (2017) in their studies observed the evolution of DPs, which also affected the software architecture. They examined the aging process in a few successful open source and industrial systems, and found evidence of pattern decay resulting from *grime* — "non-pattern-related" code.

While several studies found DPs to have a positive impact on quality, other works reached the opposite conclusions, particularly concerning maintainability, understandability and reusability. For example, Ng et al. (2012) analyzed DP as a factor that impacts the maintenance effort. They observed that maintainers who had prior in-depth knowledge of the program and the presence of solutions independent of the implemented DPs needed less time to complete maintenance tasks. In addition, pattern-unrelated solutions appeared to be more efficient and quicker. Similar observations were reported by Prechelt et al. (2001). They appreciated the flexibility offered by patterns, but also found that alternative solutions were less defect-prone or required lower maintenance effort. The results presented by Khomh and Guéhéneuc (2008) also indicated that some DPs (Composite, Abstract Factory and Flyweight) have a negative impact on the reusability and understandability of code. Additionally, Feitosa et al. (2019) reported that classes not involved in DPs, or involved in some complex and change-prone patterns, e.g., Decorator and Template Method, can be more prone to violations. This was also confirmed by Wendorff (2001) who analyzed a large commercial project and found that uncontrolled use of patterns resulted in severe maintenance problems that could not be easily resolved.

Finally, the question of the stability of DP-related code has yet to be definitely answered. Ampatzoglou et al. (2015) conducted a change-impact analysis on 65.000 Java open-source classes. The results indicate that classes that play exactly one role in a DP are more stable than classes with more roles or those not involved in any DP.

The presented works clearly indicate that despite the prevailing recommendations concerning DPs, using them is not a universal remedy for various issues in software development. This may suggest the presence of latent variables that affect the observed results.

2.3. Empirical studies on architectural smells

ASs are a relatively new concept, without a fully established terminology: they are also called *antipatterns* (Khomh et al., 2011), *design smells* (Ganesh et al., 2013), or *architectural flaws* (Mo et al., 2015). Some authors use these terms interchangeably, while others propose more complex ontologies, e.g., antipatterns can be considered as a subtype of architectural smells, and some of them, like Tangle or Hub, are called antipatterns (Khomh et al., 2011) or architectural smells (Suryanarayana et al., 2014; Garcia et al., 2009). To date, there are only few empirical studies on AS. This could be attributed to the limited availability of tools for detecting AS, with only a few notable exceptions, e.g., Azadi et al. (2019).

The hypothesis of increased change- and defect-proneness resulting from the presence of AS was observed by Mo et al. (2015), who proposed five file-level architectural flaws and found their significant correlation with error-proneness and change-proneness. Similarly, Oyetoyan et al. (2015) identified a positive correlation between the presence of Cyclic Dependency AS and the change frequency.

Other authors have analyzed the impact of AS on maintainability. For instance, Le et al. (2018) analyzed the relationships between AS and issues reported in *issue trackers* (e.g., Jira). They found that ASs have tangible negative consequences, resulting in implementation issues and increased maintenance effort.

Herold (2020) investigated the relationship between the presence of AS and manually validated *architectural violations*. He identified the links for Unstable Dependencies and for Hub-Like Dependencies ASs, but with small effect sizes. He concluded that the presence of AS cannot alone explain erosion of architecture, but it does play a contributory role. Brunet et al. (2012) also studied the evolution of architectural violations in 76 versions of four systems, by comparing the intended and recovered architectures of a system. They found that architectural violations tend to intensify as software evolves, and usually *a few design entities are responsible for the majority of violations.*

The impact of AS on system understanding was highlighted in an empirical study by Abbes et al. (2011). They found that the presence of several antipatterns in one code entity significantly impedes the programming performance of developers.

The comprehensive and diverse nature of AS has also been the subject of investigation. Sas et al. (2019) analyzed how instability-related ASs evolve in time, by analyzing three different types of ASs and mining their characteristics from 524 versions in 14 software projects. Fontana et al. (2019b) presented a study on possible correlations between AS and CS. They appeared to be linked only in a few cases. Therefore, the presence of AS cannot be inferred from CS and they require separate methods for dealing with them.

Prioritizing ASs was a subject for Martini et al. (2018), who validated their presence in four industrial projects, based on the refactoring effort reported by project developers. Vidal et al. (2016) proposed a suite of three scoring criteria for *code smells* to pursue prioritization. They concluded that the criteria helped developers to locate symptoms of *architectural problems*. Finally, Fontana et al. (2019a) developed various machine learning models trained on multiple versions of four Java projects. They found that the presence of AS in a previous versions of the system also affects them in the future.

The presented works indicate that the presence of AS affects various important properties and quality characteristics of software systems, and that DPs can also deliver diverse results. These observations provide a preliminary justification for the conjectured impact of AS on DP, which requires a more thorough analysis.

3. Empirical study design

In the following section, we describe our research questions which guide the study of AS and DP relationship. In particular, we aim to *(1)* investigate the frequency of both phenomena in the analyzed projects, and *(2)* understand whether there are specific pairs of ASs and DPs frequently involved in a relationship.

Our study aims to answer the following research questions:

RQ1: *What is the distribution and prevalence of ASs and DPs in Java projects?* We want to understand how many components (class or package) are affected by ASs and DPs. This is useful for researchers and developers to have an overview about the frequency of the two phenomena, and also for us to set the stage for the other research questions. In particular, we consider two specific issues:

> *RQ1.1*: *Is there a difference in the distribution of ASs and DPs with respect to the considered projects?* Rationale: We aim to analyze how both ASs and DPs are distributed in the considered software projects. Our aim is to identify the most frequent types of smells in software projects, so that the developers can focus their attention on them.

RQ1.2: *Is there a difference in the distribution of ASs and DPs with respect to various application domains?* Rationale: This question helps in assessing to what extent specific application domains are affected by ASs and DPs. It is important to know which domain is more open to ASs, so that developers can be more aware of ASs when dealing with projects belonging to it. It is important also to compare the presence of ASs with the presence of DPs, to understand if domains interested by many DPs are also the ones with fewer ASs.

RQ2: *Which DP-AS pairs display significant relationships?* Rationale: A strong relationship among a specific type of AS and DP may indicate that the implementation of a pattern is a root cause for the introduction of a smell, meaning that those DPs, contrary to their purpose, have a negative impact on quality. However, we could discover that some ASs imply the presence of some DPs. This could mean that those ASs are false positives i.e., the smell is present because developers had intended to program it. On the other hand, it could signify that certain DPs can help in mitigating the negative impact caused by the ASs, and they are employed by developers to remedy the problem.

RQ3: *Can the presence of ASs imply the absence of DPs?* or viceversa *Can the presence of DPs imply the absence of ASs?* Rationale: The presence of the considered ASs in the system affects the dependency structure of the system itself. DPs are implemented by manipulating the structure of the system and the implementation is possible when the structure respects some principles (e.g. it must be acyclical). Hence we want to investigate if the presence of specific ASs results in the absence of specific DPs, and viceversa. This would reinforce the conclusion that ASs and DPs are mutually exclusive concepts.

3.1. Analyzed projects and collected data

In this study we analyzed the 60 open source Java projects presented in Table 1. This is a subset of projects being curated under Qualitas Corpus (QC) (Tempero et al., 2010), and their selection was conditioned by the availability of properly compiled code, which is necessary for the ASs detector. For the projects, we report their application domain (Domain), name (Project), analyzed version (Version), number of classes (NOC), number of packages (NOP), and the total number of lines of code (TLOC). These projects have diverse characteristics: they are assigned to seven different domains (Graphics, Database, IDE, Middleware, Parser, Testing, Tool), have different sizes (ranging from 2809 to 651 118 TLOC), and are developed by different open source communities (Apache, Eclipse, etc.). In order to balance the dataset, the original domains defined in QC have been adjusted: 3D/Graphics/Media, Diagram Generator/Data Visualization and Games have been named "Graphic", and "SDK projects" were merged with "IDE" as "IDE".

3.1.1. Architectural smells

We detect and analyze the following ASs (Arcelli Fontana et al., 2016):

- *Unstable Dependency (UD)*, which describes a subsystem (component) dependent on other subsystems that are less stable than itself. This may cause a ripple effect of changes in the system. Instability of a component is measured with the metric proposed by Martin (1995) as the ratio of outgoing dependencies to the total number of dependencies of the component. UD is detected at the package level.
- *Hub-Like Dependency (HL)*, which arises when an abstraction has (both outgoing and incoming) dependencies with a large number of other abstractions. HL is detected in classes and packages.
- *Cyclic Dependency (CD)*, which refers to a subsystem (component) that is involved in a chain of relations breaking the desirable acyclic nature of a subsystem's dependency structure. It is hard to release, maintain or reuse the subsystems involved in a dependency cycle in isolation. CD is detected for classes and packages.

We focus on these three ASs since they are based on dependency issues: dependencies are of great importance in software architecture and components that are highly coupled and with a high number of dependencies are considered more critical, since they have higher maintenance costs. They represent relevant sources of architectural debt (Martini et al., 2018) and are detected by Arcan. Moreover, they are suitable for representation in the dependency dataset (see Section 3.3) and the AS-affected dependencies can be easily compared with the DP-affected dependencies.

3.1.2. Design patterns

We collected the data on all the DPs described in Table 2. These patterns have been defined by GoF (Gamma et al., 1995), except for *Proxy2*, which is a variation of the Proxy pattern. In that case, also called *Dynamically-Typed Proxy* (Binun and Kniesel, 2012), the Proxy role has an association to Subject role (named subject) and the method `Request()` declared in Proxy invokes an abstract method having the same signature through the Subject association. We chose the reported patterns since they are all identified by one tool, called *Pattern4*, and represent a different subset of patterns proposed in the GOF catalog (Gamma et al., 1995).

We performed our analysis on different aggregations of data: project data, application domain data, and the entire dataset. By the "*granularity level*" we mean the specific type of a Java component: class or package.

3.2. Tools

To detect DPs we used **Pattern4** (Tsantalis et al., 2006), capable of extracting the patterns from the analysis of Java project's static structure. In particular, the implemented detection methodology is based on similarity scoring between graph vertices, where a graph represents the project under analysis. We decided to use this tool due to its free availability and the large number of detected DPs. Moreover, it has been validated on 3 open source projects, having very high (95%–100%) precision and recall (Tsantalis et al., 2006).

To detect AS, we employed **Arcan** (Arcelli Fontana et al., 2017), capable of detecting five ASs in Java code. In this work we consider only three ASs, since the detection of *Implicit Cross Package Dependency* (ICPD) is based on the history of a project's revisions, and *Specification–Implementation Violation* (SIV) requires information about the initially intended architecture of a system, which is not available. Arcan analyzes Java projects and represents them as dependency graphs, where classes and packages are nodes and their dependencies are edges. The tool exploits graph databases to perform graph queries, which allows for higher scalability in the detection and management of different kinds of dependencies. The detailed detection techniques for the ASs have been described in Arcelli Fontana et al. (2017, 2016). We decided to use Arcan for AS detection as only a few tools are freely available for this task (Azadi et al., 2019). The results of the detection have been validated on ten open source projects (Arcelli Fontana et al., 2016) and on two industrial projects, demonstrating a high precision value of 100% and 63% of recall (Arcelli Fontana et al., 2017). Moreover, the results reported by Arcan were also positively validated on the feedback provided by practitioners working on four (Martini et al., 2018) and one (Fontana et al., 2020) industrial projects.

Table 1
Analyzed projects.

Domain	Project	Version	NOC	NOP	TLOC
Database	axion	1.0-M2	257	13	24 163
	cayenne	3.0.1	2 991	185	192 431
	db-derby	10.9.1.0	3 010	217	651 118
	hsqldb	2.0.0	644	26	143 870
	squirrel-sql	3.1.2	73	2	6 944
	hibernate	4.2.0	7 119	856	431 693
Graphic	batik	1.7	2 299	81	178 469
	displaytag	1.2	320	32	20 498
	drawswf	1.2.9	311	34	27 674
	itext	5.0.3	583	34	78 348
	jasperreports	3.7.4	1 709	61	169 821
	jext	5.0	761	59	60 160
	marauroa	3.8.1	247	41	17 733
	megamek	0.35.18	1 859	37	242 836
IDE	checkstyle	5.6	533	42	36 641
	colt	1.2.0	381	24	35 919
	drjava	stable-20100913-r5387	1 210	30	89 477
	eclipse SDK	3.7.1	24 871	1425	2 484 311
	jpf	1.5.1	140	10	13 342
	nakedobjects	4.0.0	2 975	496	133 936
	trove	2.1.0	72	4	5 845
Middleware	informa	0.7.0	223	26	13 874
	jena	2.6.3	1 279	48	65 774
	jspwiki	2.8.4	582	70	60 250
	jtopen	9.4	1 915	15	342 032
	openjms	0.7.7-beta-1	616	66	39 435
	oscache	2.3	115	22	7 624
	picocontainer	2.10.2	206	15	9 253
	xmojo	5.0.0	22	9	2 809
	quartz	1.8.3	269	51	28 557
	QuickServer	2.1.0	196	28	18 339
	sunflow	0.07.2	209	22	21 970
	tapestry	5.1.0.5	2 119	139	97 206
Parser	ant	1.8.2	1 608	122	127 507
	antlr	3.4	381	20	47 443
	apache-maven	3.0.5	837	143	65 685
	javacc	5.0	107	8	14 633
	jparse	0.96	75	4	24 796
	nekohtml	1.9.14	64	7	7 647
	xalan	2.7.1	1 402	86	183 709
	xerces	2.10.0	947	53	125 973
Testing	cobertura	1.9.4.1	160	34	54 555
	emma	2.0.5312	290	27	21 492
	findbugs	1.3.9	1 432	67	110 782
	fitjava	1.1	95	5	3 457
	jmeter	2.5.1	1 038	175	94 778
	junit	4.10	171	28	6 580
	log4j	2.0-beta	606	61	32 658
	pmd	4.2.5	872	88	60 739
Tool	freecs	1.3.20111225	146	12	22 645
	heritrix	1.14.4	656	48	64 916
	james	2.2.0	306	31	27 087
	jfreechart	1.0.13	1 037	69	143 062
	jgraph	5.13.0.0	298	34	31 818
	jgraphpad	5.10.0.2	375	22	24 208
	jmoney	0.4.4	83	4	8 197
	jsXe	04_beta	251	14	18 494
	pooka	3.0–080505	491	28	44 474
	proguard	4.9	648	35	62 618
	webmail	0.7.10	115	19	10 147

3.3. Dataset

We now introduce the dataset used during the analysis, named *Dependency Dataset*. It has been created in response to the observation that in object-oriented projects both ASs and DPs affect the structure of the code, specifically the dependencies between classes/packages. Moreover, for DPs, the direction of the dependencies and their type (class dependency, inheritance dependency and interface dependencies) are important for defining the pattern itself. Arcan is able to represent both classes/packages (nodes) and dependencies (edges) in the dependency graph. Hence, the dataset built and exploited for this work is *edge-based*.[1]

With the **Dependency dataset** we are able to model each dependency in the dependency graph and determine if the dependency is part of an AS and/or a DP. The dependencies that Arcan can extract belong to one of two granularity levels: class

[1] Replication package is available at https://drive.google.com/drive/folders/1ONSTAwyvK9d7gGp70kgLXfDZvkU80xz1.

Table 2
Detected design patterns.

Name	Type	Description
Factory Method (FM)	Creational	Define an interface for creating an object, but let subclasses decide which class to instantiate.
Prototype (P)	Creational	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
Singleton (S)	Creational	Ensure a class only has one instance, and provide a global point of access to it.
Object Adaptor (A)	Structural	Convert the interface of a class into another interface clients expect.
Composite (C)	Structural	Compose objects into tree structures to represent part-whole hierarchies.
Decorator (D)	Structural	Attach additional responsibilities to an object dynamically.
Bridge (B)	Structural	Decouple an abstraction from its implementation so that the two can vary independently.
Proxy (PR)	Structural	Provide a surrogate or placeholder for another object to control access to it.
Proxy2 (PR2)	Structural	Proxy variation reported by Gunter Kniesel and Alex Binun from University of Bonn
Command (COM)	Behavioral	Encapsulate a request as an object thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Observer (O)	Behavioral	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
State (ST)	Behavioral	Allow an object to alter its behavior when its internal state changes.
Strategy (STR)	Behavioral	Define a family of algorithms, encapsulate each one, and make them interchangeable.
Template Method (TM)	Behavioral	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
Visitor (V)	Behavioral	Represent an operation to be performed on the elements of an object structure.
Chain of Responsibility (COR)	Behavioral	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.

Table 3
(Class) dependency dataset features.

head	tail	type	weight	AS1	AS2	...	AS#	DP1	DP2	...	DP#

or package. There are four types of dependencies considered: *classDependency* between class A and class B, if there is at least a method call from A to B; *inheritanceDependency* between class A and class B, if A extends B; *implementationDependency*, between class A and interface B, if A is an implementation of B; *packageDependency*, between package C and package D, if a class in C has a (class/inheritance/implementation) dependency with a class in D.

The *Class dependencies* dataset, whose features are reported in Table 3, provides information about:

- the *head* of the edge, i.e., the name of the class from which the dependency originates;
- the *tail* of the edge, i.e, the name of the class at which the dependency ends;
- the *type* of dependency. Three types are considered: *classDependency*, *inheritanceDependency* and *implementationDependency*.
- the *weight* of the edge (the number of times the dependency is realized in the code);
- the smells (*AS1 ... AS#*) and the design patterns (*DP1 ... DP#*) that involve the dependency. ASs and DPs are considered binary features (we count the presence/absence of ASs and DPs in the dependency).

The *package dependencies* dataset stores the same features except for the "type"; as for packages we consider only *packageDependency*. Moreover, the detected AS for classes are (Class) Cyclic Dependency and (Class) Hub-Like Dependency; the detected smells for packages are (Package) Cyclic Dependency, (Package) Hub-Like Dependency and Unstable Dependency. Since DPs are structures implemented at the class level, we had to aggregate our data at the package level. We say that a package dependency is involved in a DP, if the corresponding dependency at the class level (i.e., a cross-package class dependency) is involved (see Fig. 1). For instance, if package A depends on package B (A → B) and class C1 that belongs to A depends on class C2 of package B (C1 → C2), then all the DPs involving (C1 → C2) are also counted for (A → B).

This dataset allows for counting:

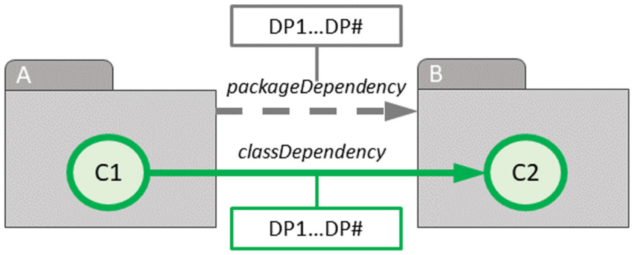

Fig. 1. Aggregation of DP from class level to package level.

- The number of dependencies involved in at least one AS,
- The number of dependencies involved in at least one DP,
- The number of dependencies involved in an AS and a DP,
- The number of dependencies not involved in any AS,
- The number of dependencies not involved in any DP,
- The number of dependencies not involved in any AS or DP.

3.4. Analysis of the dependency dataset

As explained in Section 3.3, our dataset consists of two levels: the **(1) class level**, with *data point*: Java class dependency, and *features*: DP and AS data; **(2) package level**, with *data point*: Java package dependency, and *features*: DP data of the package dependencies, computed as described in Section 3.3, and AS data. We consider (Class) Hub-Like Dependency and (Class) Cyclic Dependency smells, (Package) Hub-Like Dependency, (Package) Cyclic Dependency and (Package) Unstable Dependency smells.

Given this representation, we used three analysis techniques in order to answer our RQs related to the frequency of ASs and DPs and the relationships among them.

Comparison of DP and AS frequencies. In order to answer *RQ1.1 and RQ1.2*, we computed the absolute and relative frequencies of both ASs and DPs at the class and package level.

Used technique: distribution statistics.

Correlation analysis. In order to answer *RQ2 and RQ3* we performed the Spearman (1904) and Kendall and Gibbons (1990) correlation analysis on the entire dataset. Correlation analysis is a method of statistical evaluation used to study a relationship between two variables. Spearman *rho* and Kendall *tau* rank correlations are two of the correlation coefficients commonly used to measure the strength of the relationship between two variables that are not normally distributed. We chose them because we checked the normality of our variables and discovered they were not normal. In particular, we ran the Anderson–Darling test (Nelson, 1998) for normality. We could not use the well known Shapiro–Wilk test (Shapiro and Wilk, 1965) because our variables exceeded the maximum number of data-points (>5000) allowed in input by the test. Moreover, we could not use the Kolmogorov–Smirnov test, since it is not suitable when estimating the parameters from the data (as we do), and it also expects a continuous distribution that does not contain any ties (repeated values). Instead, the Anderson–Darling test does not require the mean and the standard deviation to be supplied. Additionally, we exploited Q–Q plots (Wilk and Gnanadesikan, 1968). A Q–Q plot is a graphical method for comparing two probability distributions by plotting their quantiles against each other. These plots are often used when the dataset is large enough to introduce bias in the Shapiro–Wilk test, as in our case.

Used technique: computation of *Spearman's rho* and *Kendall's tau* rank correlation coefficients. *Used tool*: R language `cor()` function.[2]

Association rules extraction. Moreover, related to answer *RQ2 and RQ3* we aim to exploit association rules to identify relationships among ASs and DPs. An association rule is the expression of a relationship among data items in a dataset. It is an *if-then* statement composed by an `{antecedent}` and a `{consequent}`. An example of a rule for the AS and DP dataset is `{HL, Singleton}` → `{CD}`, which can be read as "*If* Hub-Like Dependency and Singleton pattern affect a dependency at the same time, *then* the dependency belongs to a cycle". The association rule extraction technique aims to automatically extract such rules from a dataset composed of *transactions*; in our case a transaction corresponds to a vector of the binary features (AS and DP) associated to a single dependency.

We also adopt commonly used metrics for evaluating a quality of a rule: support, confidence (Agrawal et al., 1996), conviction and lift (Brin et al., 1997), with the following definitions.

Given a rule defined as $X \to Y$,

Support (*Supp*) of a rule is the ratio of transactions that match the rule with respect to the entire dataset.

Confidence (*Conf*) is the ratio of transactions that contain both the antecedent X and the consequent Y.

$$Conf(X \to Y) = \frac{Supp(X \cup Y)}{Supp(X)} \tag{1}$$

Lift is the ratio of the observed support to the expected support, if X and Y are independent. If lift is equal to 1, it means that the rule is not significant for the dataset.

$$Lift(X \to Y) = \frac{Supp(X \cup Y)}{Supp(X) \times Supp(Y)} \tag{2}$$

Conviction (*Conv*) is the ratio of the probability that X will appear without Y if they are dependent, divided by the observed frequency of the appearance of X without Y. Conviction is useful to measure the degree of implication of the association, i.e., how much Y depends on X; in particular, high conviction indicates that the consequent is highly dependent on the antecedent, while conviction of value 1 means that the items are unrelated.

$$Conv(X \to Y) = \frac{1 - Supp(Y)}{1 - Conf(X \to Y)} \tag{3}$$

Used technique: an implementation of the Apriori algorithm (Agrawal and Srikant, 1994). *Used tool*: `apriori` function from the `arule`[3] R package. *Parameters*: for all datasets, we fixed the *minimum support* to 0.001 and we reported the rules with *confidence* ≥ 0.6, as used in other empirical studies (Walter et al., 2018). In the following, we provide the details concerning the three analyses conducted on both classes and packages.

Table 4
Descriptive statistics for the dependency dataset.

	Total	min	max	mean	median	st. dev.
class dependencies						
# dependencies	226 066	44	32 834	3767.767	1692	5467.270
#not AS/DP	139 416 (62%)	27	21 054	2323.600	863.5	3574.855
#AS	43 420 (19%)	8	4 687	723.667	266	998.051
#DP	31 883 (14%)	0	7 578	531.383	240	1083.069
#AS/DP	86 650 (38%)	17	11 780	1444.167	571	2058.908
package dependencies						
# dependencies	17 362	1	4 450	289.367	102	685.826
#not AS/DP	5341 (30%)	0	1 860	89.017	23	276.486
#AS	6036 (34%)	0	1 101	100.600	65	455.039
#DP	2084 (12%)	0	804	34.733	8	114.867
#AS/DP	12 021 (69%)	0	2 590	200.350	79	413.653

4. Results

This section reports the results of our analysis starting from (1) the computation of statistical information, to answer RQ1.1 and RQ1.2, followed by (2) correlation analysis, and (3) association analysis to answer RQ2 and RQ3. The complete results can be found in the replication package.

4.1. AS and DP distribution and prevalence results

Starting with the answer to RQ1, we provide a description of the results regarding how much the different types of ASs and DPs affect the analyzed projects at the two studied granularity levels.

Table 4 reports the frequency of ASs and DPs at class and package level, without considering their types. Tables 5 and 6 report the results for ASs and DPs, respectively, at the class and package level, depending on the different types of ASs and DPs. The tables indicate data extracted from all the 60 projects, and for each reported quantity we provide the *total*, the *minimum*, the *maximum*, *mean* and *median* values. In particular, Table 4 shows some aggregated statistics for the entire dataset. Each row reports the number of analyzed dependencies (#dep.), the number of dependencies not involved in any ASs or DPs (#not AS-DP), the number of dependencies *only* affected by ASs (#AS), the number of dependencies *only* involved in DPs (#DP) and, finally, the number of dependencies involved in ASs *or* DPs (#AS-DP). The percentage values reported in the parenthesis of Table 4 are in relation to the total number of dependencies. The percentages reported in Tables 5 and 6 do not sum up to 100%, because some dependencies are affected by more than one smell or design pattern at the same time.

In the following, we introduce a detailed analysis of our dataset by (1) investigating the frequency of ASs and DPs in the 60 Java projects and their domains, (2) reporting the ASs statistics and (3) the DPs statistics.

2 https://www.r-project.org/.

3 https://cran.r-project.org/web/packages/arules/index.html.

Table 5
Statistics for architectural smells in the dependency dataset.

AS	Total	min	max	mean	median	st. dev.
class dependencies						
CD	51 959 (94%)	4	5504	865.983	281.5	1213.474
HL	8301 (15%)	0	1477	138.35	63.5	240.8923
package dependencies						
CD	8445 (84%)	0	1627	140.750	63	266.9363
HL	2780 (27%)	0	259	46.333	29	55.8397
UD	4818 (48%)	0	907	80.300	36	152.6016

4.1.1. Comparison of ASs and DPs frequencies

Figs. 2 and 3 show the frequency of ASs and DPs in the subject projects at the class and package granularity levels. The x-axis indicates the projects ordered by ascending number of dependencies, and the y-axis shows the number of ASs and DPs for each project (highlighted in two different colors). As we can see from the diagram, the trend is growing for both ASs and DPs.

By inspecting the proportion of ASs and DPs with respect to the different project domains (Figs. 4 and 5), it becomes clear that the projects that present high disparity in the proportion of ASs and DPs belong to the *Database* domain. In particular, the number of dependencies affected by ASs at the class level is 10 900, while by DPs it is 15 712; 3927 and 8570 at package level. Through Figs. 6 and 7 we can understand which specific DP is most prevalent, along with the highest concentration of ASs and in which domain. In general Template Method, Singleton and Factory Method are the most frequent DPs, at both class and package level and in each domain. However, ASs are more concentrated in selected domains, such as Parser and Testing at class level, Graphic and Parser at package level. By merging together the two indicators, DP presence and AS concentration, the following patterns are the most frequent, with the highest concentration of smells: Template Method, Singleton and Factory Method in Parser and Testing domains at class level; Template Method, Singleton and Factory Method in Database and IDE domains at package level.

4.1.2. Architectural smells statistics

Concerning RQ1.1, and in particular the frequency ASs, Table 5 reports the frequency of ASs in the 60 analyzed projects, considering their type (*CD*, *HL*, *UD*) and granularity level (class and package). In particular, we indicate the total, minimum, maximum and mean number of the instances of each specific type. The most frequent smell at both class and package level is CD. It is also the only smell which is present in all projects: in fact, the minimum value of detected CD is 4, meaning that there is at least one project with at least 4 cycles. At the package level, HL is less diffused than CD.

Concerning RQ1.2, we now report the diffuseness of ASs in relation to the application domains of the analyzed projects.

Figs. 8 and 9 show the frequency of class and package ASs over the 7 different domains. The most diffused smell, for both the granularity levels, is Cyclic Dependency (CD). At the class level, its presence is almost equal in all the domains, with $mean = 7422$. The most affected domain is Database, with 10 149 CD instances. However, the number of CD instances varies depending on the different domains at package level. Graphic, Middleware and Parser domains have the lowest number of CD instances, while the Testing and Tool domains have a medium number of the smell and, finally, Database and IDE are strongly affected by CD, with more than 1500 smells.

Concerning Hub-Like Dependency (HL), the most affected domain at the class level is IDE, while at the package level it is Middleware. For the Unstable Dependency (UD) smell, Database is the most affected domain, with 1246 instances.

4.1.3. Design pattern statistics

Still regarding RQ1.1, Table 6 present the collected statistics of the detected DPs, respectively at class and package level, for each type. The most diffused DPs are Template Method (18 518 class dependencies, 7915 package dependencies), Singleton (13 915 class dependencies, 7393 package dependencies) and Factory Method (11 768 class dependencies, 8035 package dependencies). Adaptor pattern was the only one to remain undetected during the analysis. In general, the most frequently detected patterns belong to the *creational* category, while the *structural* category contains the lowest number of patterns. With regard to the applications domains, we did not find a specific prevailing DP (see Fig. 10 and Fig. 11).

4.2. Results of the correlation analysis

To answer RQ2, we tested the correlation between AS and DP through the computation of Spearman *rho* and Kendall *tau* correlation coefficients.

Before running the test, we checked the normality of our variables with the Anderson–Darling test (Nelson, 1998). The null hypothesis is that the data are normally distributed; the alternative hypothesis is that the data are non-normal. We set the significance level at 0.05. We ran the test and rejected the null hypothesis for all the considered AS and DP variables, at both class and package level. We also generated Q−Q plots to confirm the results of the test, and they gave the same result. The scripts and the results of the normality tests can be found in the replication package.

The coefficient values are in the range $[-0.058, 0.134]$ for class dependencies, and $[-0.013, 0.117]$ for package dependencies.

As for the Spearman's analysis, the Kendall correlation was tested between all the possible pairs of AS and DP. The coefficient values are in the range $[-0.009, 0.019]$ for class dependencies, and $[0, 0.040]$ for package dependencies.

Since all the coefficient values are very close to 0 **(no correlation), we can conclude that this analysis did not discover an interesting relationship among ASs and DPs.**

For this reason, we do not report the result tables, but they can be consulted in the replication package.

4.3. The results of mining the association rules

In order to answer RQ2 and RQ3, we now present the results obtained from the association rule analysis, performed on both *class* and *package* dependency datasets. For each rule, we report their *antecedent* (Left Hand Side, LHS), their *consequent* (Right Hand Side, RHS) and commonly used metrics that evaluate its quality: support, confidence (Agrawal et al., 1996), conviction and lift (Brin et al., 1997).

As explained in Section 3.4, we collected the rules with minimum support of 0.001 and confidence greater than 0.6.

Some of the rules contain only ASs. This happens because we ran the rules extraction on all the dependency datasets, and some dependencies are affected at the same time by different smell types, but by no DP. For the sake of completeness, we leave such rules in our tables, even if we do not discuss them.

4.3.1. Class rules

Table 7 reports 4 rules extracted from the (class) dependency dataset. Their support is in the range $[0.001, 0.024]$ and the conviction in the range $[2.070, 4.034]$. The involved ASs are Hub-Like Dependency (HL) and Cyclic Dependency (CD), while among the DPs there are Singleton (S), Factory Method (FM) and Template Method (TM). CD smell is the consequence of all the rules,

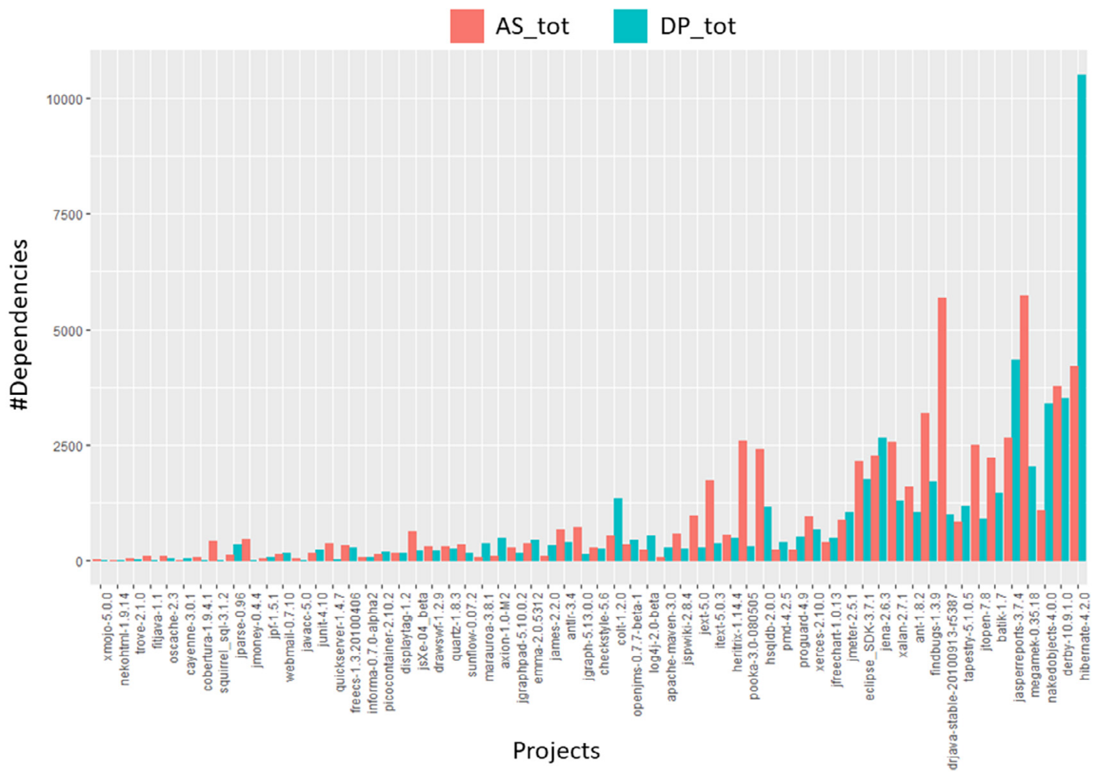

Fig. 2. Frequency of class level AS and DP in 60 Java projects.

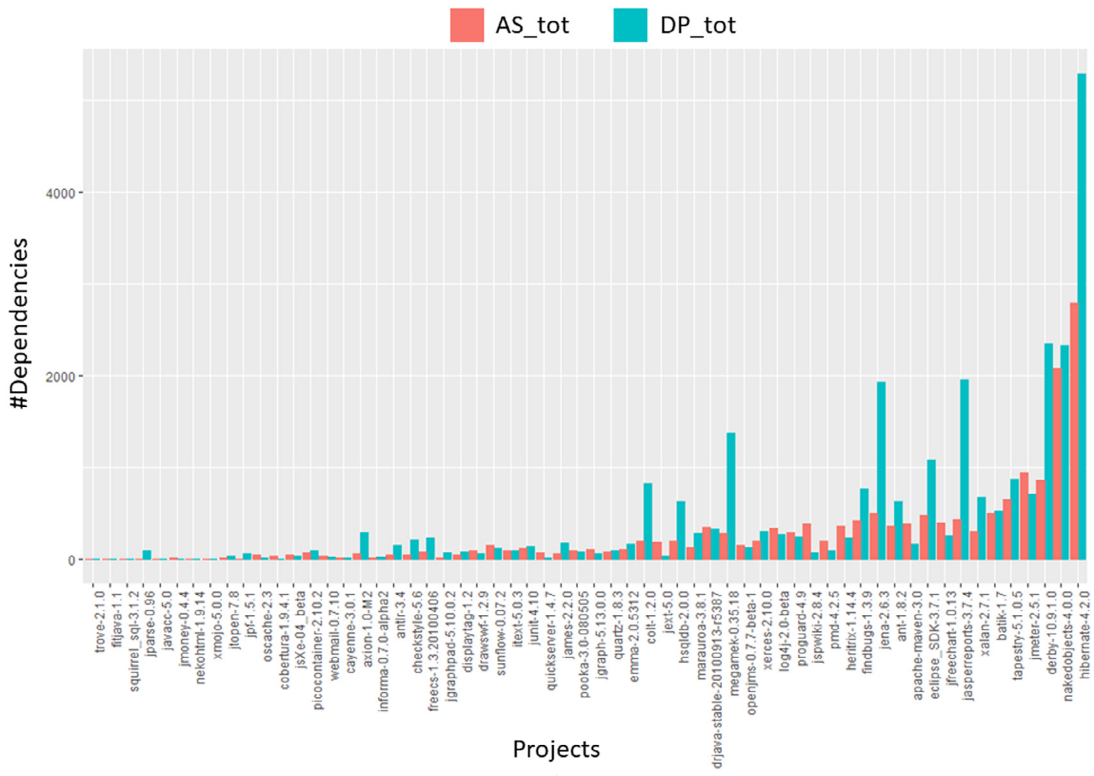

Fig. 3. Frequency of package level AS and DP in 60 Java projects.

apart from one where a TM pattern is present. This set is not surprising, since these are the most frequent types of AS and DP (see Section 4.1).

We conducted a manual validation of the 4 rules, by inspecting the code of the classes that match the rules. This was helpful for providing an interpretation of the discovered rules and to answer *RQ2* and *RQ3*. For each rule we provide *(1) its description*,

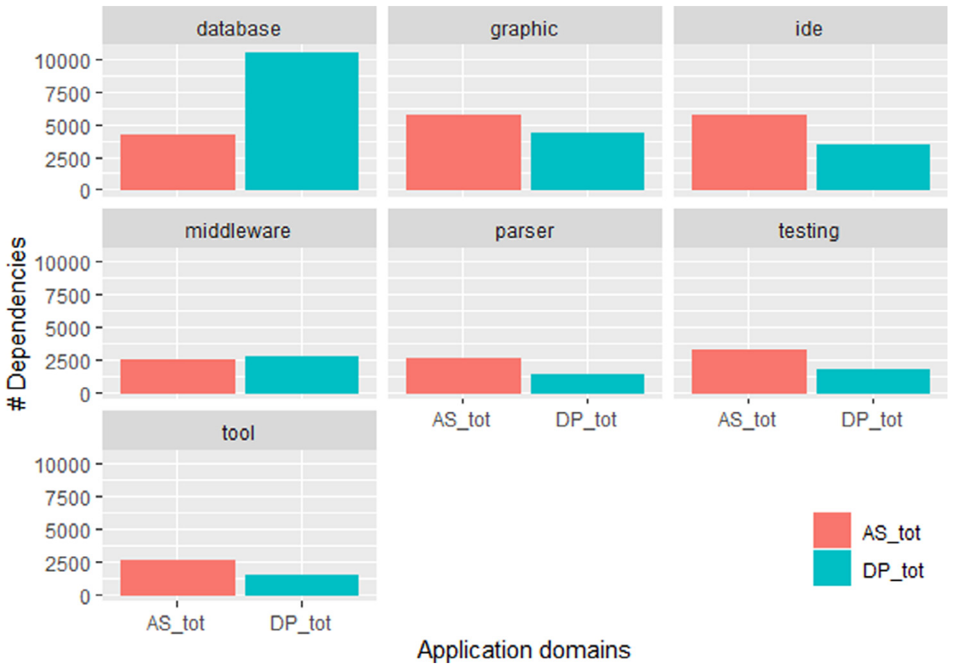

Fig. 4. Frequency of ASs and DPs in 7 domains — Class level.

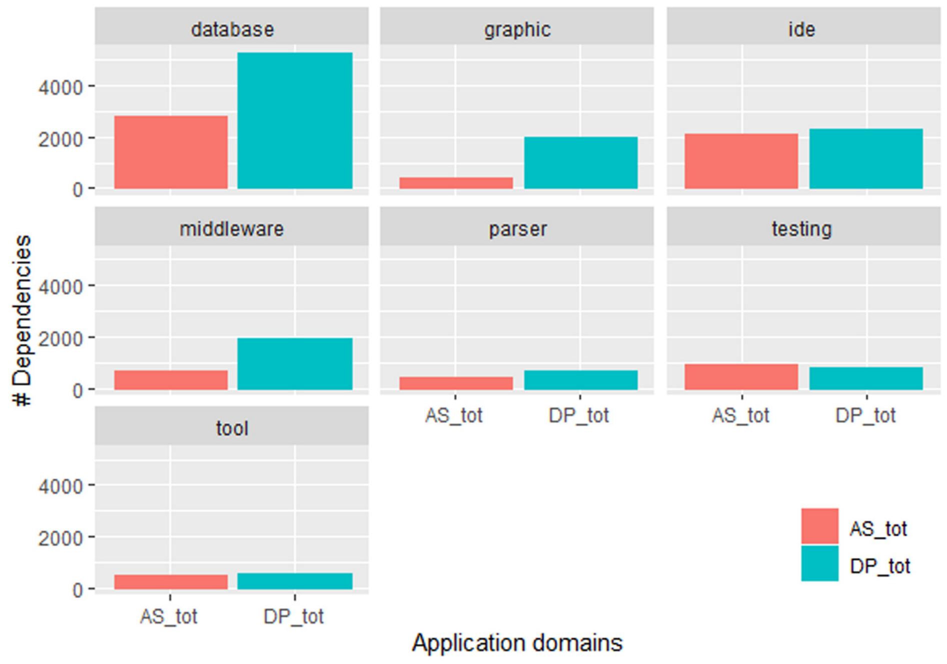

Fig. 5. Frequency of ASs and DPs in 7 domains — Package level.

to understand how to read the rule, *(2) a matching example*, found in the analyzed projects, and *(3) interpretation*.

We need to emphasize that the reported rules refer only to the subject dataset, and should not be freely extrapolated to other data. Additionally, they may not be applicable to all analyzed dependencies; in Table 7 column "Support" indicates the prevalence of the rule in the dataset and "Confidence" shows how frequently the consequent is collocated with the antecedent.

However, thanks to the association rule mining, we found that in some cases the presence of an AS is linked to the presence of specific DPs. The aim of this interpretation is to provide practitioners and researchers with useful hints on what to do when specific combinations of AS and/or DP appear. For instance, we found some examples of false positive AS and cases of the potentially unsafe use of DP, e.g., when they are likely to introduce a new AS into the system.

In the following, we present an analysis of each class rule.

(R1)	{HL, Singleton} $\rightarrow$ {CD}

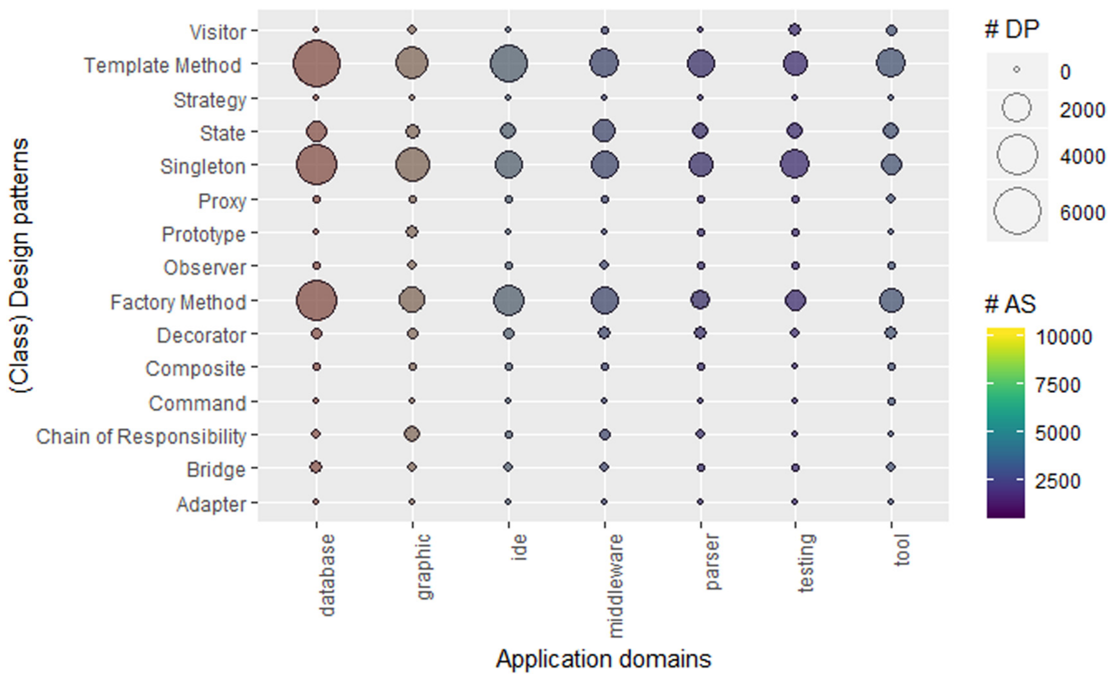

Fig. 6. Frequency of ASs and DPs in 7 domains — Class level.

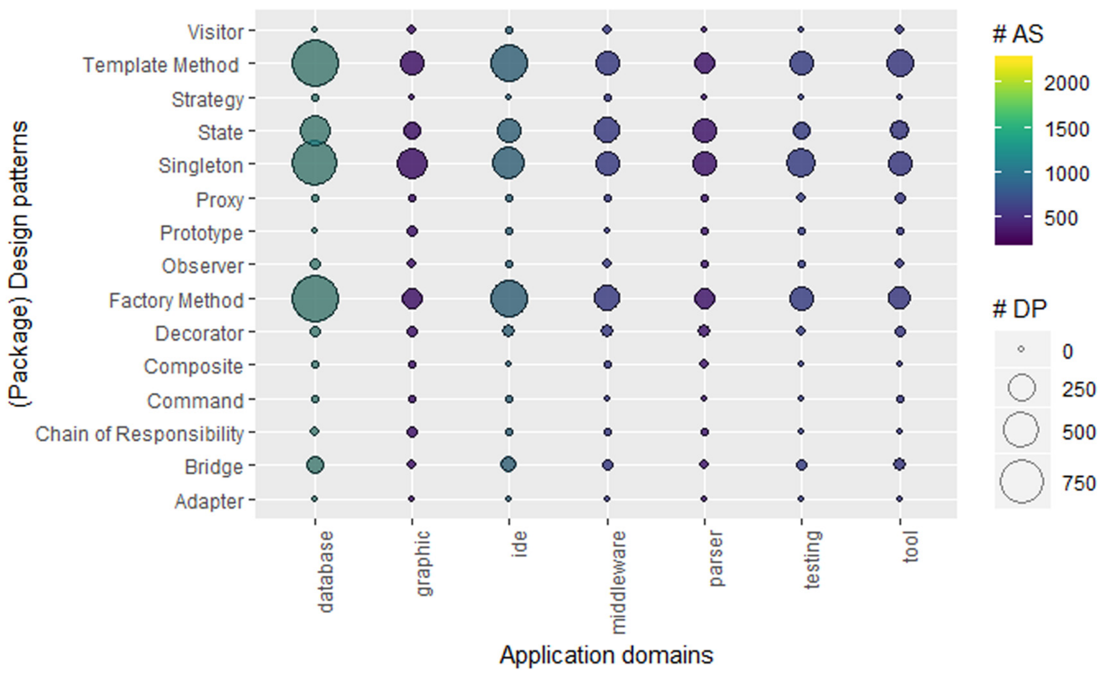

Fig. 7. Frequency of ASs and DPs in 7 domains — Package level.

Description: if a dependency is affected by Hub-Like Dependency and Singleton, then this dependency also belongs to a cycle.

Example: from the manual validation of the projects from which such a rule was extracted, we identified two scenarios:

1. One of the two classes involved in the dependency is both an HL and a Singleton class.
2. One of the two classes involved in the dependency is a HL, while the other class is a Singleton.

This happens because our dataset considers dependencies at the expense of classes/packages, without the information about which of them is affected by a smell or a DP. With regard to *case 1*, we report the example of the Findbugs project. One of its packages named `edu.umd.cs.findbugs.gui2`, which groups the classes employed to build a graphical interface, contains a class named "MainFrame" which is a large HL (FanIn = 96, FanOut = 111) and a Singleton. This class has many Cyclic Dependencies with other classes from the GUI. The reason is that when the GUI frame creates a new listener (i.e., a new instance of a GUI class), it passes itself in the constructor to enable callbacks (Verhoeff,

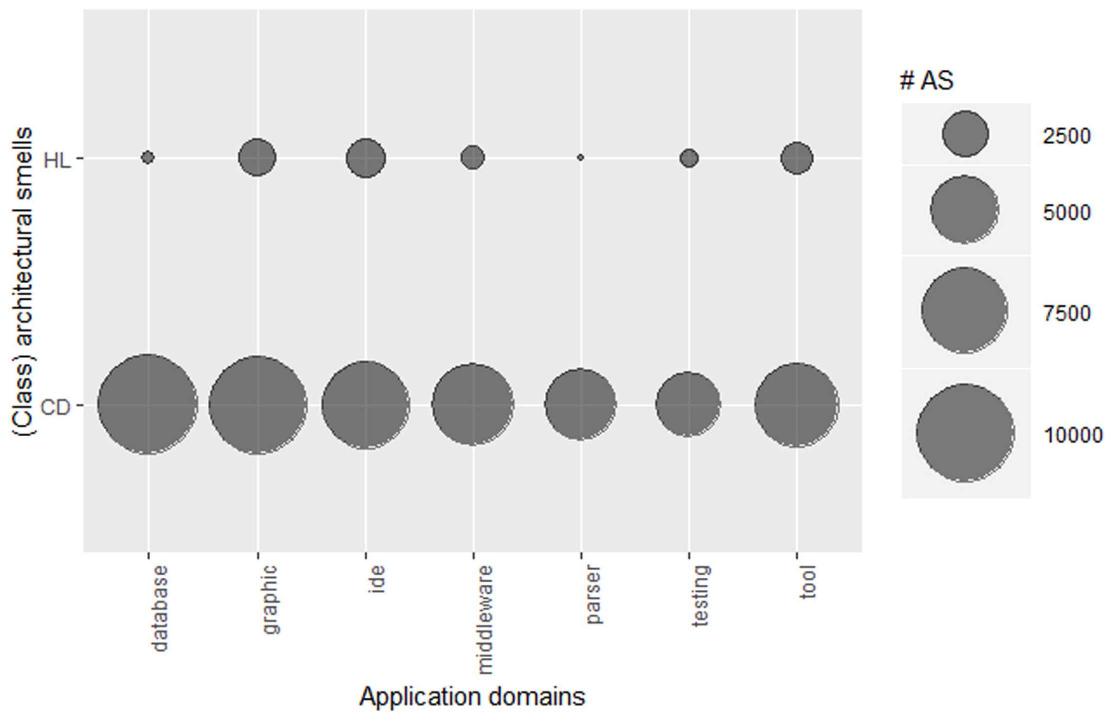

Fig. 8. Frequency of ASs in 7 domains — Class level.

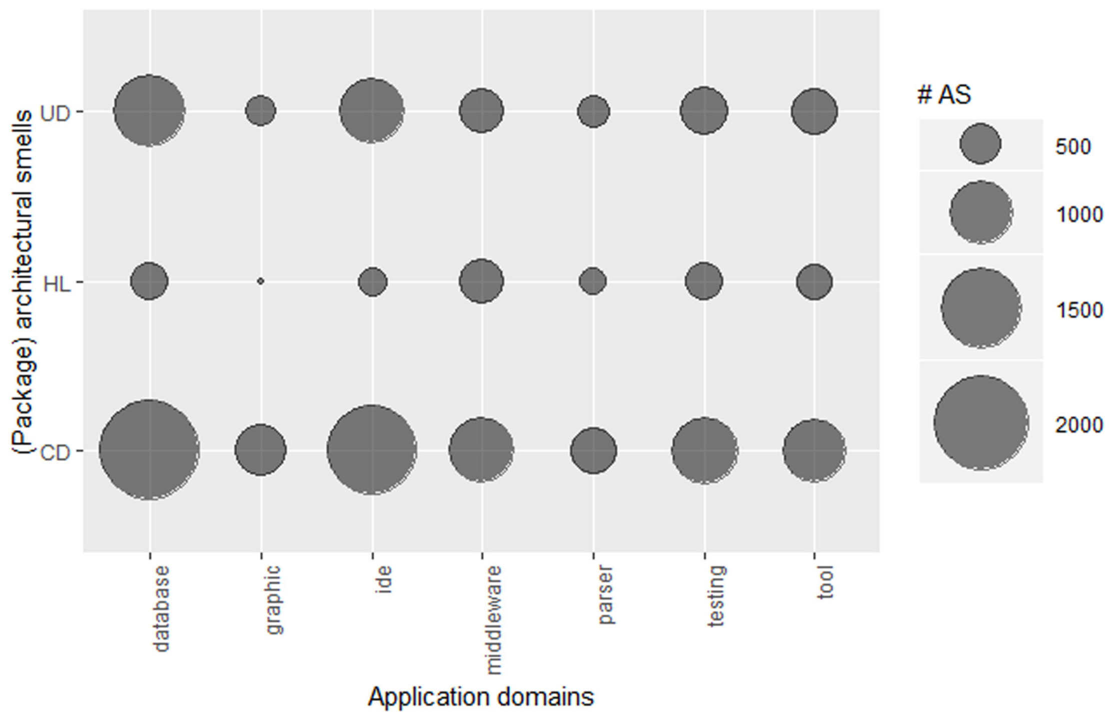

Fig. 9. Frequency of ASs in 7 domains — Package level.

2012). With regard to *case 2*, we report an example from the project antlr-3.4. `org.antlr.tool.Grammar`, which is an HL and depends on the Singleton class `org.antlr.misc. IntervalSet`. The dependencies between them form a cycle.

Interpretation: classes like "MainFrame" could be examples of false positives of the AS instances, where developers introduce Cyclic Dependencies in the project in order to implement a callback. Alternatively, this could be a case of improper implementation of the callback, which introduces a unique class which manages both the creation of the listeners and their callbacks. The fact that the class owns too much responsibility explains the presence of the Hub-Like Dependency smell, at the same time the presence of many Cyclic Dependency smells is caused by the callbacks.

On the other hand, classes such as "Grammar" may be a symptom of the overlapping of a manager/core class (the hub) with the use of the Singleton class: it is neither a false positive, nor a bad implementation of the pattern; instead, it could be

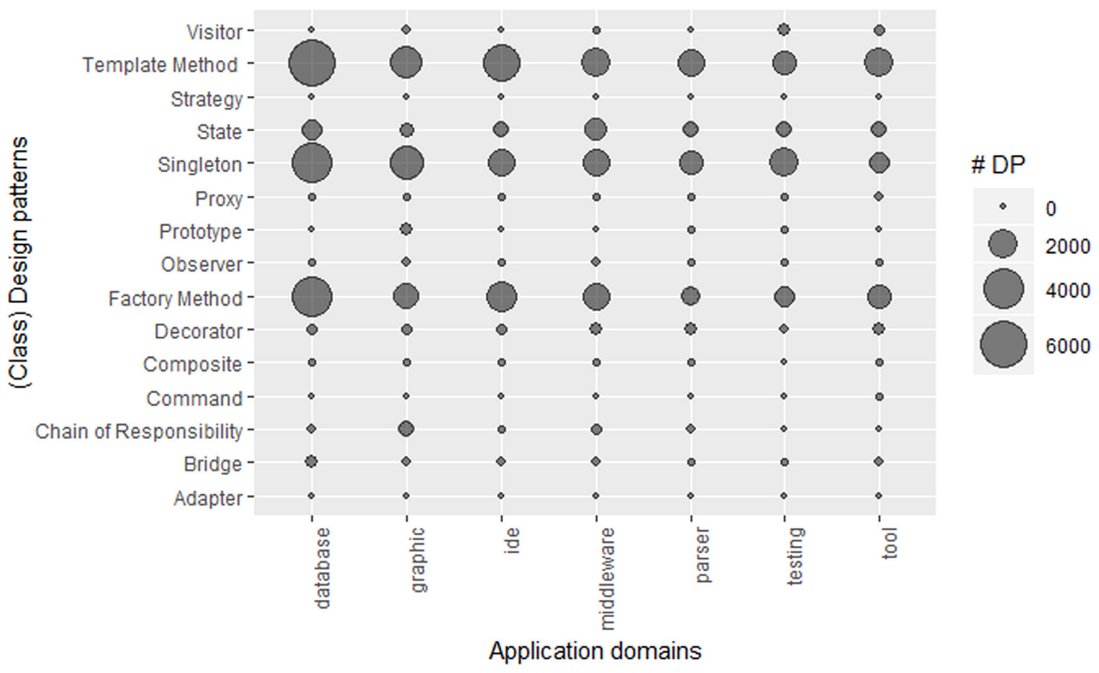

Fig. 10. Frequency of DPs in 7 domains — Class level.

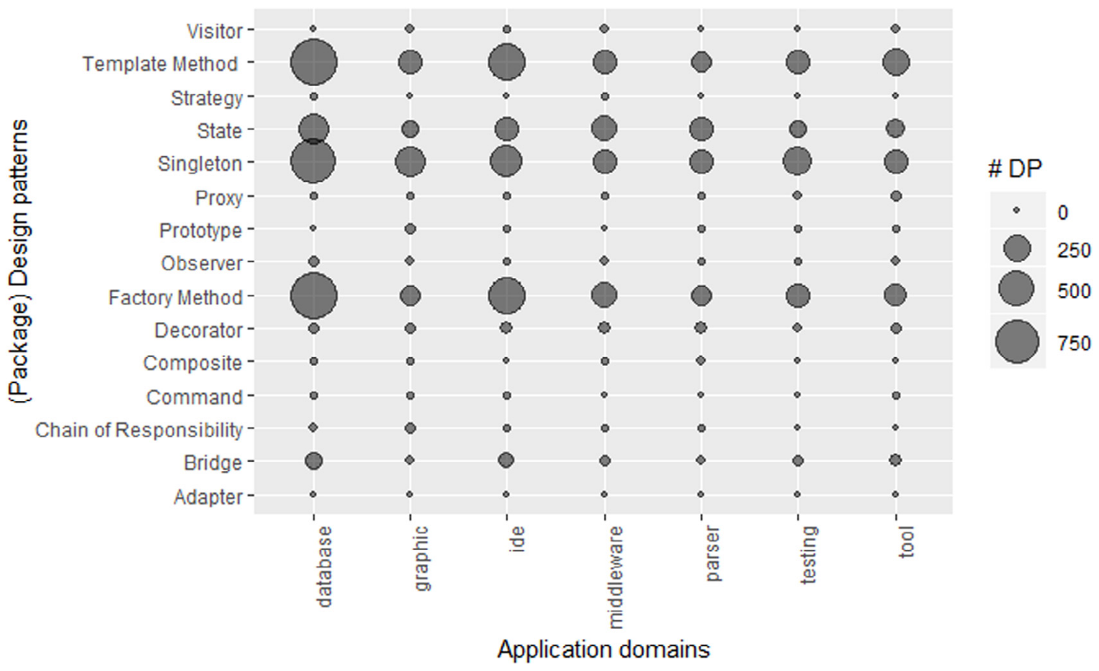

Fig. 11. Frequency of DPs in 7 domains — Package level.

a particular *type* of Hub-Like Dependency. The fact that such dependencies are also implied in CD smells strengthens the idea that the hub is a collector of dependencies and centralizes the activity of the system. We found many classes matching this rule, hence this information could be helpful for refining the definition of ASs and proposing a new classification for them.

(R2) {HL, Template Method} → {CD}

Description: if a dependency is affected by Hub-Like Dependency and Template Method, then such a dependency also belongs to a cycle.

Example: in project Emma, a class `Attribute_info` in `com.vladium.jcd.cls.attribute` is both an HL and an AbstractClass of TemplateMethod. Source `FileAttribute_info` extends `Attribute_info`, but the latter in some cases returns instances of the former (Broken Hierarchy Suryanarayana et al., 2014). `SourceFileAttribute_info` overrides the template method.

Table 6
Dependency dataset — design pattern statistics.

Design pattern	# Dependencies (class)					# Dependencies (package)				
	Total	%	min	max	mean	Total	%	min	max	mean
Factory method	11 768	27%	0	2447	196.133	2245	37%	0	1741	133.917
Prototype	157	0.3%	0	126	2.617	25	0.40%	0	124	2.35
Singleton	13 915	32%	0	2992	231.917	2372	39%	0	1394	123.217
Adaptor	0	0%	0	0	0	0	0%	0	0	0
Command	20	0.04%	0	7	0.333	8	0.13%	0	3	0.133
Composite	94	0.2%	0	14	1.567	11	0.18%	0	6	0.45
Decorator	867	%2%	0	174	14.45	140	2%	0	74	6.683
Observer	133	0.3%	0	34	2.217	45	0.75%	0	10	0.917
State	3 458	7%	0	551	57.633	1144	19%	0	438	37.85
Strategy	8	0.01%	0	4	0.133	3	0.05%	0	2	0.05
Bridge	421	0.9%	0	105	7.017	200	3%	0	61	4.517
Template method	18 518	42%	0	4449	308.633	2355	39%	0	1807	131.917
Visitor	319	0.7%	0	135	5.317	22	0.36%	0	68	2.017
Proxy	162	0.3%	0	22	2.7	32	0.53%	0	7	0.7
Chain of responsibility	623	1%	0	356	10.383	33	0.55%	0	113	2.417

Interpretation: this is a false positive of Hub-Like Dependency, because dependencies toward the abstract class of Template Method affected by HL are actually resolved to its concrete classes. This rule is useful for refining the detection of the Hub-Like Dependency smell.

(R3) {CD, Factory Method} → {Template Method}

Description: if a dependency is affected by Cyclic Dependency and Factory Method, then such a dependency also belongs to the Template Method DP.

Example: In the Derby project, belonging to the Database domain, many classes are *creators*, i.e., implement the Factory Method pattern. As an example, let us consider the class `Connection`, which is a core element in the management of a new database connection. This class is responsible for creating the class `Agent` and, at the same time, it forms a cycle with it. Moreover, it also implements the Template Method pattern, since other classes (e.g., `NetConnection`) inherit from this class and extend its template methods.

Interpretation: From the manual validation of the project matching the rule, it became apparent that classes involved in Factory Method are likely to be part of the same cycle. This is justified by the fact that newly created classes often need to use the creator class. Moreover, Factory Method classes tend to also implement the Template Method pattern. However, we did not find improper use of the two patterns. Hence, developers should only pay attention when using Factory Methods, because their use may lead to the introduction of new cycles.

(R4) {HL} → {CD}

We do not provide an in-depth explanation of this rule, because it does not include a DP in its body, and because the relationship it describes has already been investigated for rule (R1) and (R2). In brief, we suggest that the relationship between HL and CD is justified by the fact that classes affected by HL are by definition involved in many dependencies, some of which can also be cyclic ones. Since the role of hub classes is usually central in the system, it is reasonable that other classes referred by hubs call back the hubs themselves. We find the discussion of the rules with DPs more interesting, since when DPs are combined with the presence of these two smells it gives us additional actionable information (e.g., about false positives).

4.3.2. Package rules

Fig. 12 shows how the conviction and support of the rules change depending on the *order* of the rules. We define the order

Table 7
Association rules at class level.

LHS		RHS	Support	Confidence	Lift	Conviction
{HL, S}	→	{CD}	0.001	0.809	3.520	4.034
{CD, FM}	→	{TM}	0.004	0.628	7.676	2.473
{HL}	→	{CD}	0.024	0.661	2.879	2.276
{HL, TM}	→	{CD}	0.002	0.628	2.732	2.070

of a rule as the number of distinct ASs and DPs appearing in the body of the rule (Walter et al., 2018).

As we can see from the plot, the rules with the highest conviction are the ones where order is equal to 2 and 3, hence we report all the rules with the order up to 3. The results of the package rule extraction are reported in Table 8. The Table shows the first 50 rules, ordered by conviction, with order $\leq$ 3. The total number of rules is 188; they are documented in the replication package. Support is in the range [0.001, 0.233] and confidence in the range [0.610, 0.954].

We manually validated the rule with the highest conviction value, which puts the Visitor DP and the Cyclic Dependency AS into a relationship.

(R5) {V} → {CD}

Description: if a dependency involves Visitor, then such a dependency is also involved in a Cyclic Dependency.

Example: The packages of the projects matching this rule appear to break the Visitor pattern in different packages, causing dependencies to spread from one package to the other, resulting in Cyclic Dependencies. An example is in the Eclipse project, where the packages `org.eclipse.core.internal.resources` and `org.eclipse.core.resources` contain the implementation of the pattern and are involved in a Cyclic Dependency. This is due to the anonymous class `ResourceTree$1` in package org.eclipse.core.internal.resources which implements the Visitor interface named `IResourceVisitor`, located in package `org.eclipse.core.resources`. The circular dependency appears because the package `org.eclipse.core.resources` contains some classes (example: `ResourcesPlugin` and `WorkspaceJob`) which depend on classes belonging to a different package named `org.eclipse.core.internal.resources`.

Interpretation: When the *concrete visitors* of the Visitor pattern are located in a different package than the Visitor interface, it is more likely that a Cyclic Dependency will occur between the two packages. This is due to the split of the elements of the DP into different packages. Developers should pay attention when they implement this pattern, in order to avoid the introduction of Cyclic Dependencies.

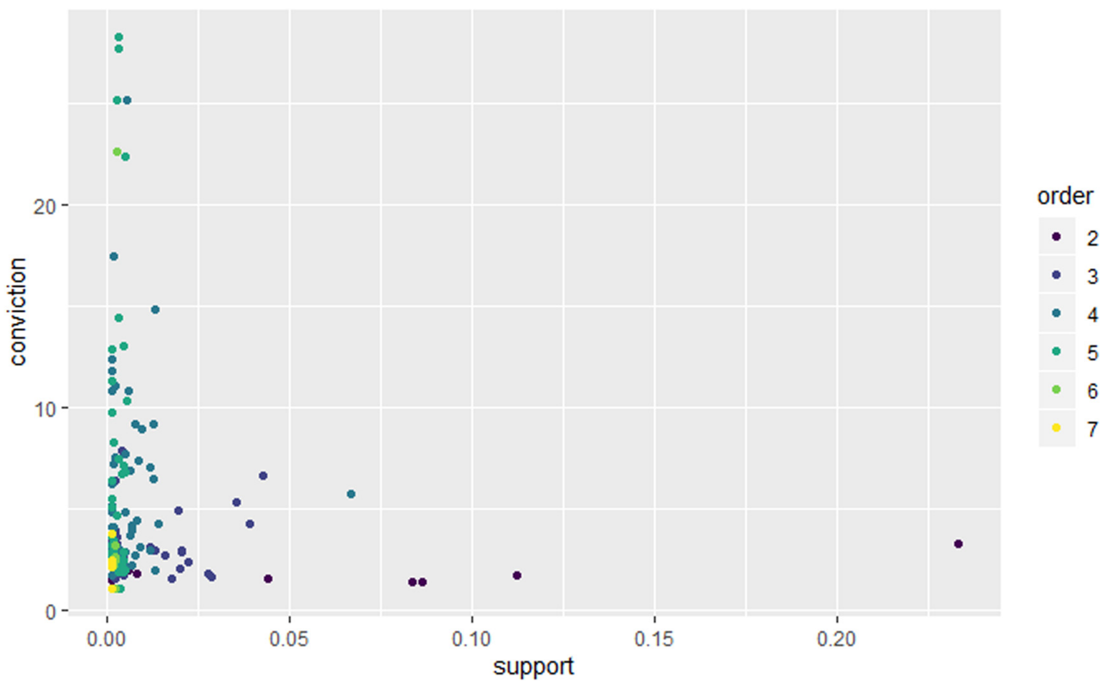

Fig. 12. The order of package association rules.

5. Discussion

In this section, we provide the answers to each RQ and discuss the obtained results.

RQ1: what is the distribution and prevalence of ASs and DPs in Java projects? ASs affect 24% of the class dependencies dataset and 57% of package dependencies. The most diffused type of AS is Cyclic Dependency, which affects 29.75% of the analyzed dependencies. *Megamek* is the project with the highest number of CDs, with 5504 occurrences. With regard to the Hub-Like Dependency smell, the most affected project is *DrJava*.

DPs cover 19% of the dataset class dependencies and 34% of package dependencies: the most frequent DP is Template Method with 18 518 instances at the class level (42%), and 2355 instances at the package level (39%). Hibernate is the project with the highest number of DP instances, in particular Template Method (4449 instances), Singleton (2992 instances), Factory Method (2447 instances) at class level; and Template Method (677 instances, %15), Singleton (680 instances, %15), Factory Method (609 instances, 13%) at package level.

We also counted the architecture dependencies which are involved at the same time in ASs and DPs (the number of examples where ASs and DPs are collocated). With respect to the total number of class dependencies which make up our dataset, this intersection represents 5%, and at package level it reaches 23%. This means that, at least at the package level, the collocation concerns almost 1/4 of the dependencies, which also makes it pertinent for qualitative investigation.

Within this question we also addressed the following topics:

RQ1.1: Is there a difference in the distribution of ASs and DPs with respect to the considered projects?

In general, both ASs and DPs grow in number as projects grow in size, where by size we mean the number of dependencies (Figs. 2 and 3). Given that, we notice that the trend oscillates for both ASs and DPs. For instance, in correspondence with the interval of projects [*jspwiki*, *hslqdb*] (see the *x*-axis in Fig. 2, on the right) the ASs show a clear increase at the expense of the number of DPs, while in slightly larger projects we observe a decrease of ASs and an increase of DPs. This effect occurs along all the graphics and could indicate that projects with a large number of DPs have fewer ASs.

However, this aspect goes over the aim of our research questions and should be investigated further in order to reach a clearer conclusion, for instance by increasing the number of analyzed projects and by testing the correlation between the number of ASs and DPs of the projects.

RQ1.2: is there a difference in the distribution of ASs and DPs with respect to various application domains? At the class level, the most AS-affected application domains are Database and IDE, specifically by Cyclic Dependency (which is also the most frequent smell in general) and Hub-Like Dependency. The same happens at the package level, where Database and IDE are strongly affected by Cyclic Dependency (Database from Unstable Dependency too), and Middleware domain is the most affected by Hub-Like Dependency.

In general, without considering the granularity, Graphics is the domain with the highest number of ASs (in particular Cyclic Dependency), while the majority of the DPs are present in the Database application domain. Hence, from the analysis conducted on our dataset, we can conclude that in general ASs and Dps are equally frequent in the different domains, with a unique exception, the Database domain, where the number of DPs far exceeds the number of ASs.

RQ2: Which design pattern-architectural smell pairs display significant relationships? As reported in RQ1, we found, especially at the package level, a share of dependencies where ASs and DPs are collocated. Hence, we analyzed whether there is a relationship between specific types of ASs and DPs. In terms of a correlation coefficient, our analysis did not identify any significant relationships. The values of Spearman and Kendall coefficients are close to 0, indicating no correlation for any of the analyzed pairs of AS and DP. However, the results from mining the association rules are more interesting. In particular, we extracted 4 rules regarding class dependencies that relate Hub-Like Dependency and Cyclic Dependency smells with Template Method, Factory Method and Singleton patterns.

We also identified some examples of false positive ASs by manually validating these rules. We found that code suspected of

Table 8
Association rules at package level (top 50).

LHS		RHS	Support	Confidence	Lift	Conviction
{V}	→	{CD}	0.001	0.955	1.962	11.299
{P}	→	{CD}	0.001	0.840	1.727	3.210
{COR}	→	{CD}	0.001	0.636	1.308	1.412
{O}	→	{CD}	0.002	0.756	1.553	2.101
{D}	→	{CD}	0.006	0.729	1.498	1.892
{B}	→	{CD}	0.008	0.710	1.460	1.771
{ST}	→	{CD}	0.044	0.667	1.371	1.542
{S}	→	{CD}	0.086	0.630	1.296	1.389
{TM}	→	{CD}	0.083	0.614	1.262	1.331
{HL}	→	{CD}	0.112	0.700	1.438	1.710
{UD}	→	{CD}	0.233	0.840	1.726	3.205
{O,ST}	→	{CD}	0.001	0.800	1.645	2.568
{D,ST}	→	{CD}	0.002	0.655	1.346	1.487
{S,D}	→	{FM}	0.001	0.769	5.949	3.773
{FM,D}	→	{CD}	0.002	0.782	1.607	2.354
{S,D}	→	{CD}	0.001	0.846	1.740	3.338
{D,TM}	→	{CD}	0.002	0.870	1.788	3.938
{HL,D}	→	{CD}	0.002	0.846	1.740	3.338
{UD,D}	→	{CD}	0.002	0.919	1.889	6.334
{ST,B}	→	{FM}	0.003	0.636	4.921	2.394
{ST,B}	→	{TM}	0.003	0.610	4.500	2.219
{ST,B}	→	{CD}	0.003	0.753	1.549	2.081
{S,B}	→	{FM}	0.002	0.680	5.259	2.721
{FM,B}	→	{TM}	0.004	0.673	4.960	2.641
{B,TM}	→	{FM}	0.004	0.679	5.250	2.712
{FM,B}	→	{CD}	0.005	0.727	1.495	1.883
{S,B}	→	{TM}	0.002	0.700	5.161	2.881
{S,B}	→	{CD}	0.002	0.840	1.727	3.210
{B,TM}	→	{CD}	0.004	0.697	1.433	1.696
{HL,B}	→	{CD}	0.002	0.857	1.762	3.595
{UD,B}	→	{CD}	0.004	0.934	1.921	7.807
{FM,ST}	→	{CD}	0.018	0.661	1.359	1.514
{S,ST}	→	{CD}	0.011	0.833	1.712	3.069
{ST,TM}	→	{CD}	0.013	0.825	1.696	2.932
{HL,ST}	→	{CD}	0.016	0.808	1.662	2.680
{UD,ST}	→	{CD}	0.020	0.895	1.839	4.879
{FM,S}	→	{CD}	0.020	0.742	1.525	1.990
{FM,TM}	→	{CD}	0.028	0.676	1.389	1.584
{HL,FM}	→	{CD}	0.020	0.818	1.681	2.815
{UD,FM}	→	{CD}	0.035	0.904	1.858	5.333
{S,TM}	→	{CD}	0.027	0.712	1.464	1.783
{HL,S}	→	{CD}	0.020	0.823	1.692	2.899
{UD,S}	→	{CD}	0.042	0.922	1.896	6.619
{HL,TM}	→	{CD}	0.022	0.783	1.610	2.369
{UD,TM}	→	{CD}	0.039	0.879	1.808	4.252
{HL,UD}	→	{CD}	0.067	0.910	1.870	5.694

the Cyclic Dependencies AS can be intentionally implemented inside a *callback*, which is similar to the Observer pattern (Verhoeff, 2012). With regard to the Hub-Like Dependency, we realized that when it is combined with Template Method, the smell is actually a false positive. This finding is particularly useful for refining the detection methods applicable to ASs.

Moreover, we were able to extract 188 rules at the package level. We tried to provide an interpretation of the resulting rules by manually reviewing the code of the analyzed projects. We did this for all the rules at the class level and for one rule at the package level. By manual validation we identified cases where a DP led to the presence of a specific smell. For instance, from the analysis of a package rule *we discovered that the implementation of Visitor pattern that is spread across multiple packages is more likely to introduce Cyclic Dependencies among the packages*. As outlined in Section 2, some patterns have also been found to be defect-prone in other studies: Observer and Singleton (Vokac, 2004), Composite, Prototype, and Adaptor-Command (Aversano et al., 2009). In particular Sousa et al. (2019) found the Adaptor-Command pattern to be highly correlated with code smells, which are usually indicated as the counterpart of AS at code level, i.e., symptoms of poor software quality. Although the results reported in the literature are not fully consistent, they indicate that some patterns appear more troublesome than others. Our results seem to partially confirm these findings and can be useful to developers, who should pay attention when implementing DPs that could be associated to ASs.

RQ3*: *Can the presence of architectural smells imply the absence of design patterns?* or vice-versa *Can the presence of design patterns imply the absence of architectural smells? We counted the number of dependencies where only ASs are present, without DPs: at the class level they comprise 19% of the dataset, while at the package level 34%. On the other hand, dependencies involved only in DP are 14% at class level and 12% at package level. Given that the AS and DP collocation is 5% (class) and 23% (package), we could say that, on the basis of the numbers, there are more cases where the two concepts are mutually exclusive, i.e., more examples where the presence of one of the two excludes the other.

6. Threats to validity

In this section, we discuss threats to the validity of our study, following the structure suggested by Yin (2009).

Threats to **construct validity**, which concern the identification of the measures adopted, can occur due to errors in the data extraction and preparation phases. Moreover, we build and rely on a dataset based on object-oriented dependencies: there could be errors in the construction of the dataset, and we could have extracted biased results affected by the dependency representation. However, we relied on well known R libraries (e.g., `dyprl`) to manipulate our data and we manually checked our dataset and published both datasets and analysis results in the replication package.[4] Finally, some DPs are related to methods whose granularity level was not considered in this study, and we aggregated class data to obtain a representation of DPs at the package level. Hence, the analysis of the package-level DP and AS could have led to erroneous conclusions. On the other hand, we carefully explained our aggregation method and we provide examples of manual validation also for results at the package level.

Threats to **internal validity** are factors that could have affected the results obtained. In our case, they may be due to the choice of the statistical methods used for the analysis of the dependency dataset and their implementation in the used tools (R libraries and KNIME platform). We mitigate this threat by relying on multiple sources, such as similar empirical studies (Fontana et al., 2019b) conducted on code smells and DP correlations (Walter et al., 2018).

Threats to **external validity** refer to the generalization of the results beyond the original setting. They may arise from the nature of the projects used in our study. We analyzed only projects written in Java and that are publicly available. However, we partially mitigate such issues by analyzing a large number of projects (60). Another threat is related to the definition of software domains. Even though we relied on the categorization provided by the Qualitas Corpus (Tempero et al., 2010), we decided to merge some of them in order create a more balanced dataset, which could have influenced our results.

Threats to **reliability** concern the correctness of the conclusions reached in our study. We rely on two tools (Arcan and Pattern4) to extract dependency information and detect ASs and DPs in the analyzed projects. Both tools could be subject to systematic bias in the detection. Such threats are partially mitigated by the provided replication package and the fact that both tools are available, validated and can be applied to any compiled Java project. Validation of Arcan results has been performed on ten

4 https://drive.google.com/drive/folders/1ONSTAwyvK9d7gGp70kgLXfDZvkU80xz1.

open source projects (Arcelli Fontana et al., 2016) and on two industrial projects, with a high precision value of 100% in the results and 63% of recall (Arcelli Fontana et al., 2017). Moreover, the results of Arcan were validated using the feedback provided by practitioners working on four industrial projects (Martini et al., 2018). With regard to Pattern4, the tool has been validated on three open source projects (Tsantalis et al., 2006). The precision of all the examined patterns for all projects is 100%. Recall is 100% except for 2 patterns: Factory Method (%66.7, %25 and %100, for the 3 projects) and State (%95.6, %91.6 and %100).

The differences in the recall between Arcan and Pattern4 may affect the conclusions reported in the answer to RQ1, where we compared the distribution of ASs and DPs in the dataset. However, to address the tool bias, we manually cross-validated the results, which also revealed other interesting insights that have not been found by the static analysis.

We need to acknowledge that, despite our efforts, our study does not provide definitive answers, especially to RQ2 and RQ3. The qualitative analysis of association rules (RQ2) was limited to a few examples, while the answer to RQ3 relies only on quantitative data. The latter threat is mitigated once again by the large number of analyzed projects. Concerning our qualitative analysis, we reported the exact names of the classes/packages that we manually analyzed, and since we considered only public projects, our statements can be easily verified.

7. Conclusions and future developments

From the analysis of a given object-oriented software architecture, we can extract information about architectural smells (ASs) and design patterns (DPs), which affect the quality of the software in different, and seemingly opposite ways. In particular, ASs and DPs are considered as having opposite effects on the maintenance of a software system: the former are manifestations of sub-optimal design and architectural decisions that hinder software evolvability and maintainability, while the latter are verified, recommended solutions to recurring design problems. However, DPs have also been shown to be faulty in specific cases.

In this paper, we investigated whether the presence of ASs in a project *influences* or *is influenced by* the presence of DPs, under the hypothesis that the latter can sometimes have a negative impact on software quality (Sousa et al., 2019; Khomh and Guéhéneuc, 2008; Feitosa et al., 2019; Wendorff, 2001). We studied the presence of ASs and DPs in 60 open source Java projects and explored the possible relationships that may occur among specific types of AS and DP. We built a dataset with the results obtained from the execution of two static analysis tools, Arcan (for AS detection) and Pattern4 (for DP identification). The dataset is *dependency oriented*, i.e., we associated information on ASs and DPs to object-oriented dependencies, since both ASs and DPs are commonly recurring *structures* of the software architecture. We collected statistical information about the frequency of ASs and DPs and their collocation: all the analyses were conducted separately at two granularity levels, class and package, and we also studied the results in relation to the application domains of the analyzed projects. Then, we performed a correlation analysis with two different coefficients, Spearman *rho* and Kendall *tau*, in order to detect possible statistical correlations among ASs and DPs. Finally, we mined our dependency dataset to extract association rules and, consequently, possible associations among ASs and DPs.

Our results show that in our dataset there are more examples of dependencies which are involved *only* in ASs and *only* in DPs, i.e., our data seems to confirm that they are mostly mutually exclusive concepts. However, from the qualitative analysis we performed with the association rules on the collocation examples, we found hints about what happens when ASs and DPs overlap.

There are indeed some connections between the co-occurrence of specific types of ASs and DPs. Some of them are effects of AS false positives instances, for instance the Template Method can be a signal of a Hub Like false positive. However, some relationships occur because specific implementations of DPs can imply the introduction of ASs, as happens for the Visitor pattern, which can cause the introduction of Cyclic Dependencies. Both results are useful in different aspects: by studying design constructs such as DPs we can gain interesting ideas on how to enhance ASs detection; being aware of the fact that the implementation of a DP can lead to effects contrary to intentions (i.e., the introduction of *bad* design decision), knowledge of AS-DP relationships helps developers to focus their attention on specific fragments and structures. Moreover, this indicates the need to develop new tools able to spot such smells, starting from the presence of specific DPs.

The study is open to several extensions. What we described is a first investigation of the relationships between ASs and DPs. For instance, as reported in the discussion of RQ1.1, from the representation of the frequency of ASs and DPs we discovered a hint as to the possible correlation between the increase/decrease of AS and the related decrease/increase of DP. If we could demonstrate this correlation, that would mean that projects with a large number of implemented DPs are affected by a lower number of ASs. In the future, we aim to investigate this relationship further, by also exploring the evolution of the number of ASs and DPs in the projects with the support of statistical tests.

In terms of the association rules, by looking at the extracted rules it is clear that CD is implicated in the majority of them. This fact is not surprising, given that CD is the most frequent smell in the analyzed projects. Thus, it could be interesting to execute the rules extraction once again, by excluding dependencies affected by CD from the dataset. This may lead to the identification of more rules involving different smells with different DPs.

Finally, we aim to implement new false positives management in Arcan for the detection of Hub-Like Dependency and Cyclic Dependency, based on the association rules detected from the dependency dataset.

CRediT authorship contribution statement

Ilaria Pigazzini: Investigation, Formal analysis, Writing - original draft. **Francesca Arcelli Fontana:** Conceptualization, Supervision, Writing - original draft. **Bartosz Walter:** Validation, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Abbes, M., Khomh, F., Guéhéneuc, Y., Antoniol, G., 2011. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: 15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany. pp. 181–190. http://dx.doi.org/10.1109/CSMR.2011.24.

Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A.I., et al., 1996. Fast discovery of association rules. Adv. Knowl. Discov. Data Min. 12 (1), 307–328.

Agrawal, R., Srikant, R., 1994. Fast algorithms for mining association rules in large databases. In: Proceedings of the 20th International Conference on Very Large Data Bases. In: VLDB '94, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 487–499.

Alkhaeir, T., Walter, B., 2021. The effect of code smells on the relationship between design patterns and defects. IEEE Access 9, 3360–3373. http://dx.doi.org/10.1109/ACCESS.2020.3047870.

Ampatzoglou, A., Chatzigeorgiou, A., Charalampidou, S., Avgeriou, P., 2015. The effect of gof design patterns on stability: A case study. IEEE Trans. Softw. Eng. 41, http://dx.doi.org/10.1109/TSE.2015.2414917.

Arcelli Fontana, F., Pigazzini, I., Roveda, R., Tamburri, D.A., Zanoni, M., Nitto, E.D., 2017. Arcan: A tool for architectural smells detection. In: Int'L Conf. Software Architecture (ICSA 2017) Workshops. Gothenburg, Sweden, pp. 282–285. http://dx.doi.org/10.1109/ICSAW.2017.16.

Arcelli Fontana, F., Pigazzini, I., Roveda, R., Zanoni, M., 2016. Automatic detection of instability architectural smells. In: Proc. 32nd Intern. Conf. on Software Maintenance and Evolution (ICSME 2016). IEEE, Raleigh, North Carolina, USA.

Aversano, L., Cerulo, L., Di Penta, M., 2009. Relationship between design patterns defects and crosscutting concern scattering degree: an empirical study. IET Softw. 3 (5), 395–409.

Azadi, U., Fontana, F.A., Taibi, D., 2019. Architectural smells detected by tools: a catalogue proposal. In: Proceedings of the Second International Conference on Technical Debt, TechDebt@ICSE 2019, Montreal, QC, Canada, May 26-27, 2019. pp. 88–97. http://dx.doi.org/10.1109/TechDebt.2019.00027.

Binun, A., Kniesel, G., 2012. DPJF - design pattern detection with high accuracy. In: 16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012. pp. 245–254. http://dx.doi.org/10.1109/CSMR.2012.82.

Brin, S., Motwani, R., Ullman, J.D., Tsur, S., 1997. Dynamic itemset counting and implication rules for market basket data. Spec. Interest Group Manage. Data (SIGMOD Rec.) 26 (2), 255–264. http://dx.doi.org/10.1145/253262.253325.

Brunet, J., Bittencourt, R.A., Guerrero, D.S., de Figueiredo, J.C.A., 2012. On the evolutionary nature of architectural violations. In: 19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, on, Canada, October 15-18, 2012. pp. 257–266. http://dx.doi.org/10.1109/WCRE.2012.35.

Cardoso, B., Figueiredo, E., 2015. Co-occurrence of design patterns and bad smells in software systems: An exploratory study. In: Anais Principais do XI Simpósio Brasileiro de Sistemas de Informação. SBC, pp. 347–354.

Feitosa, D., Ampatzoglou, A., Avgeriou, P., Chatzigeorgiou, A., Nakagawa, E., 2019. What can violations of good practices tell about the relationship between GoF patterns and run-time quality attributes? Inf. Softw. Technol. 105, 1–16. http://dx.doi.org/10.1016/j.infsof.2018.07.014.

Feitosa, D., Avgeriou, P., Ampatzoglou, A., Nakagawa, E.Y., 2017. The evolution of design pattern grime: an industrial case study. In: International Conference on Product-Focused Software Process Improvement. Springer, pp. 165–181.

Fontana, F.A., Avgeriou, P., Pigazzini, I., Roveda, R., 2019a. A study on architectural smells prediction. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 333–337. http://dx.doi.org/10.1109/SEAA.2019.00057.

Fontana, F., Dietrich, J., Walter, B., Yamashita, A., Zanoni, M., 2016. Antipattern and code smell false positives: Preliminary conceptualization and classification. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1. pp. 609–613. http://dx.doi.org/10.1109/SANER.2016.84.

Fontana, F.A., Lenarduzzi, V., Roveda, R., Taibi, D., 2019b. Are architectural smells independent from code smells? An empirical study. J. Syst. Softw. 154, 139–156. http://dx.doi.org/10.1016/j.jss.2019.04.066.

Fontana, F.A., Locatelli, F., Pigazzini, I., Mereghetti, P., 2020. An architectural smell evaluation in an industrial context. In: The Fifteenth International Conference on Software Engineering Advances, ICSEA 2020, 18-22 October 2020, Porto, Portugal.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA.

Fowler, M., 2002. Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.

Ganesh, S.G., Sharma, T., Suryanarayana, G., 2013. Towards a principle-based classification of structural design smells. J. Object Technol. 12 (2), 1: 1–29. http://dx.doi.org/10.5381/jot.2013.12.2.a1.

Garcia, J., Popescu, D., Edwards, G., Medvidovic, N., 2009. Identifying architectural bad smells. In: 2009 13th European Conference on Software Maintenance and Reengineering. IEEE, pp. 255–258.

Herold, S., 2020. An initial study on the association between architectural smells and degradation. In: Jansen, A., Malavolta, I., Muccini, H., Ozkaya, I., Zimmermann, O. (Eds.), Software Architecture. Springer International Publishing, Cham, pp. 193–201.

Izurieta, C., Bieman, J.M., 2013. A multiple case study of design pattern decay, grime, and rot in evolving software systems. Softw. Qual. J. 21 (2), 289–323. http://dx.doi.org/10.1007/s11219-012-9175-x.

Jaafar, F., Guéhéneuc, Y., Hamel, S., Khomh, F., 2013a. Analysing anti-patterns static relationships with design patterns. In: Electronic Communications of the European Association for the Study of Science and Technology, Vol. 59. http://dx.doi.org/10.14279/tuj.eceasst.59.930.

Jaafar, F., Guéhéneuc, Y., Hamel, S., Khomh, F., 2013b. Mining the relationship between anti-patterns dependencies and fault-proneness. In: 20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013. pp. 351–360. http://dx.doi.org/10.1109/WCRE.2013.6671310.

Kendall, M., Gibbons, J., 1990. Rank Correlation Methods. In: Charles Griffin Book, E. Arnold.

Khomh, F., Guéhéneuc, Y., 2008. Do design patterns impact software quality positively? In: 12th European Conference on Software Maintenance and Reengineering, CSMR 2008, April 1-4, 2008, Athens, Greece. pp. 274–278. http://dx.doi.org/10.1109/CSMR.2008.4493325.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., Sahraoui, H., 2011. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. J. Syst. Softw. 84 (4), 559–572. http://dx.doi.org/10.1016/j.jss.2010.11.921, The Ninth Int'l Conf. Quality Software.

Lanza, M., Marinescu, R., 2006. Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer, http://dx.doi.org/10.1007/3-540-39538-5.

Le, D.M., Link, D., Shahbazian, A., Medvidovic, N., 2018. An empirical study of architectural decay in open-source software. In: 2018 IEEE International Conference on Software Architecture (ICSA). pp. 176–17609. http://dx.doi.org/10.1109/ICSA.2018.00027.

Liskov, B.H., Wing, J.M., 1994. A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16 (6), 1811–1841. http://dx.doi.org/10.1145/197320.197383.

Martin, R.C., 1995. Object oriented design quality metrics: An analysis of dependencies. ROAD 2 (3).

Martini, A., Fontana, F.A., Biaggi, A., Roveda, R., 2018. Identifying and prioritizing architectural debt through architectural smells: A case study in a large software company. In: Software Architecture - 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24-28, 2018, Proceedings. pp. 320–335. http://dx.doi.org/10.1007/978-3-030-00761-4_21.

Mo, R., Cai, Y., Kazman, R., Xiao, L., 2015. Hotspot patterns: The formal definition and automatic detection of architecture smells. In: 12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015, Montreal, QC, Canada, May 4-8, 2015. pp. 51–60. http://dx.doi.org/10.1109/WICSA.2015.12.

Moha, N., Huynh, D.-l., Guéhéneuc, Y.-G., Team, P., 2005. A taxonomy and a first study of design pattern defects. In: IEEE International Workshop on Software Technology and Engineering Practice (STEP) 2005. p. 225.

Nelson, L.S., 1998. The Anderson-Darling test for normality. J. Qual. Technol. 30 (3), 298.

Ng, T.H., Yu, Y.T., Cheung, S.C., Chan, W.K., 2012. Human and program factors affecting the maintenance of programs with deployed design patterns. Inf. Softw. Technol. 54 (1), 99–118. http://dx.doi.org/10.1016/j.infsof.2011.08.002.

Nord, R., Ozkaya, I., Kruchten, P., Gonzalez-Rojas, M., 2012. In search of a metric for managing architectural technical debt. In: Proc. 2012 Joint Working IEEE/IFIP Conf. Software Architecture (WICSA) and European Conf. Software Architecture (ECSA). IEEE, Helsinki, Finland, pp. 91–100. http://dx.doi.org/10.1109/WICSA-ECSA.212.17.

Oyetoyan, T.D., Falleri, J., Dietrich, J., Jezek, K., 2015. Circular dependencies and change-proneness: An empirical study. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). pp. 241–250.

Prechelt, L., Unger, B., Tichy, W.F., Brossler, P., Votta, L.G., 2001. A controlled experiment in maintenance: comparing design patterns to simpler solutions. IEEE Trans. Softw. Eng. 27 (12), 1134–1144. http://dx.doi.org/10.1109/32.988711.

Sas, D., Avgeriou, P., Arcelli Fontana, F., 2019. Investigating instability architectural smells evolution: An exploratory case study. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 557–567. http://dx.doi.org/10.1109/ICSME.2019.00090.

Shapiro, S.S., Wilk, M.B., 1965. An analysis of variance test for normality (complete samples). Biometrika 52 (3/4), 591–611.

Sousa, B.L., Bigonha, M.A., Ferreira, K.A., 2019. An exploratory study on cooccurrence of design patterns and bad smells using software metrics. Softw. - Pract. Exp. 49 (7), 1079–1113.

Spearman, C., 1904. The proof and measurement of association between two things. Am. J. Psychol. 15 (1), 72–101.

Suryanarayana, G., Samarthyam, G., Sharma, T., 2014. Refactoring for Software Design Smells: Managing Technical Debt, first ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

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: Proc. 17th Asia Pacific Software Eng. Conference (APSEC 2010). IEEE, Sydney, Australia, pp. 336–345. http://dx.doi.org/10.1109/APSEC.2010.46.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T., 2006. Design pattern detection using similarity scoring. IEEE Trans. Softw. Eng. 32 (11), 896–909. http://dx.doi.org/10.1109/TSE.2006.112.

Verhoeff, T., 2012. From Callbacks to Design Patterns. October.

Vidal, S., Guimaraes, E., Oizumi, W., Garcia, A., Pace, A.D., Marcos, C., 2016. Identifying architectural problems through prioritization of code smells. In: 2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS). pp. 41–50. http://dx.doi.org/10.1109/SBCARS.2016.11.

Vokac, M., 2004. Defect frequency and design patterns: An empirical study of industrial code. IEEE Trans. Softw. Eng. 30 (12), 904–917. http://dx.doi.org/10.1109/TSE.2004.99.

Walter, B., Alkhaeir, T., 2016. The relationship between design patterns and code smells: An exploratory study. Inf. Softw. Technol. 74, 127–142. http://dx.doi.org/10.1016/j.infsof.2016.02.003.

Walter, B., Fontana, F.A., Ferme, V., 2018. Code smells and their collocations: A large-scale experiment on open-source systems. J. Syst. Softw. 144, 1–21. http://dx.doi.org/10.1016/j.jss.2018.05.057.

Wendorff, P., 2001. Assessment of design patterns during software reengineering: lessons learned from a large commercial project. In: Proceedings Fifth European Conference on Software Maintenance and Reengineering. pp. 77–84. http://dx.doi.org/10.1109/CSMR.2001.914971.

Wilk, M.B., Gnanadesikan, R., 1968. Probability plotting methods for the analysis of data. Biometrika 55 (1), 1–17.

Yin, R., 2009. Case Study Research: Design and Methods. In: Applied Social Research Methods, SAGE Publications.