Università degli Studi di Salerno

Dipartimento di Informatica

Tesi di Laurea Magistrale in

Informatica

Understanding Architectural Smells

A Novel Tool and a Family of Empirical Studies

Relatori

Prof. Andrea De Lucia

Dott. Fabio Palomba

Candidato

Manuel De Stefano

Anno Accademico 2019-2020

"Troverò una bella frase o deidca"

-Manuel

Abstract

Software engineering industry is aware of how dreadful the effects of bad design decisions are on the software life cycle. Code smells, the symptoms of these wrong decisions, have been the center of a great number of both academic and industry researches. There are, though, some bad decision, which can occur at a higher level of abstraction, that could lead to even more dreadful effects: these are architectural smells. Similarly to code smells, but higher-up in the abstraction level, architectural smells are problems or sub-optimal architectural patterns or other design-level characteristics. These have received significantly less attention even though they are usually considered more critical than code smells, and harder to detect, remove, and refactor. However, they have not been completely forgotten. Both academia and industry proposed several tools able to detect this kind of flaws, but most of them are under commercial license. So, in order to overcome this obstacle, that prevents us from deepen our comprehension of these architectural problems, in this thesis a novel tool is presented. The named Brunelleschi can detect 3 types of smells at different levels of granularity, namely Cyclic Dependency, detected at both class and level package, Hub-Like Dependency, detected at class level, and Unstable Dependency, detected at package level. The tool is then validated in a family of empirical studies, aimed to better understand the diffusion and characteristics of architectural smells.

Contents

Chapter 1

Introduction

Software maintenance is known to be the longest phase of the software life cycle. Once delivered to the customer, it is necessary to proceed with continuous and periodic changes in order to preserve its usefulness. To this extent, Lehman's first law clearly explains this phenomenon: "A system must be continually adapted or it becomes progressively less satisfactory." However, software maintenance and evolution is not a simple task, as Lehman itself suggests with his second law: "As a system evolves, its complexity increases unless work is done to reduce it."

On one hand, results of a careless or not well conducted maintenance activity have emerged from a great number of studies, since Martin Fowler published his book on *refactoring* [1]. In his book, Fowler depicted a set of deign flaws and technical debts that can arise in code-bases, when sub-optimal solutions are implemented, the so-called *code smells*, which several further studies have proved to erode the code maintainability and understandability [2]. Moreover, Fowler also suggested how to *pay back* the technical debt introduced in code-bases, *i.e.*, applying refactoring. Refactoring is the process of restructuring existing code without changing its behavior, intended to improve design, structure or implementation of the software, leaving functionalities untouched.

On the other hand, other kinds of technical debt have been overlooked. Suboptimal decision, and bad choices might occur in designing the architecture of a software as well, leading to the introduction of another type of smells, *i.e.*, architectural smells. The name of these flaws is not left to choice, as it is similar to their code counterpart, just because they represent the same kind of problem, but at a higher level of abstraction. This process of deviation from the as-conceived architecture, which eventually leads to the introduction of architectural smells, is called "architectural drift" (Medvidovic *et al.* [3]). When this deviation starts to worsen and lot of sub-optimal solutions are introduced, we are facing "architectural erosion" [3]. Erosion must be dealt in early stages, otherwise it will lead eventually to the aforementioned problems. To face this, "architecture recovery" is needed [3]. An eroded architecture can not provide the most important value - as Robert C. Martin says in his book - to the stakeholders [4], *i.e.*, flexibility and testability.

Occurring at a higher level of abstraction also means that the impact of architectural smells is widely greater: for instance, a code smell could affect a finer-grained component, such as a method, a class or even a package; an architectural smell, on the other hand, could massively impact a great number of classes or modules (*e.g.*, a Cyclic Dependency). As a direct consequence of this, detecting and refactoring out an architectural smell is way harder that a code smell. Although their *dreadfulness*, architectural smells are not so popular in industry, probably because even the definition of some of them is still difficult and debated. However, as it will presented in Chapter 2, they have not been completely forgotten. Not only a catalog proposal has been presented [5], but also, automatic tools able to detect them have been developed, both in academia and industry [6, 7].

Despite these efforts, we still know too little about this issue, and the available tool are limited or released under commercial license. This is the aim that inspired and motivated this thesis. To make another step in this direction, a novel open-source tool, Brunelleschi, has been developed, and released as IntelliJ IDEA plugin. It is able to detect 3 types of smells at different level of granularity, which are Cyclic Dependency, detected at both class and package level, Hub-Like Dependency, detected at class level, and Unstable Dependency, detected at package level. Once validated, Brunelleschi has been used to conduct a set of empirical studies aimed to better understand architectural smells diffusion and characteristics. Moreover, a survey has been conducted, in order to asses the developers' awareness about this particular issue. In Chapter 2 some related work about architectural smells are explored, in particular, a definition and an overview of the most known architectural smells is presented. Chapter 3 focuses on BRUNELLESCHI, its features, its architecture, a usage scenarios and some future enhancements. In Chapter 5.1 the design of the empirical studies is defined, while in Chapter 5.2 their results and validity will be discussed. Finally, Chapter 6 wraps everything up and a presents some possible future work.

Chapter 2

Related Works

This chapter depicts the efforts that have been done so far to cope with two of the main problems regarding architectural smells: definition and detection. Section 2.1 present the theoretical issue of defining and comprehensively cataloging architectural smells, while Section 2.1 presents the practical issue of automatic detection of architectural smells in code-bases. Finally, in Section 2.3, some architectural smell related empirical studies are presented.

2.1 Architectural Smell Definition

The oldest work that can be found about architectural smells and their comprehensive definition and classification is dated back to 2006. Lippert *et al.* published a book which for the first time dealt with the problem of architectural smells in large software systems: "Refactoring in large software projects: performing complex restructurings successfully" [8]. In the third chapter of this book, they point out the concept of architectural smells for the first time, defining them as *bad smells* that occur at a higher of the system's granularity. Thus they often require more extensive refactoring to remove them. The authors also stated that architectural smells can be found on various levels: (i) In dependency graphs and inheritance hierarchies (In uses and inheritance between classes); (ii) In and between packages; (iii) In and between subsystems (bundles of packages that constitute a sufficient concept for larger systems); (iv) In and between layers;

They also specify that for larger systems, the analysis of sub-systems and layers matters more, whereas, for smaller ones, classes and packages are more critical. Throughout the rest of the third chapter, they present a long list of architectural smells grouped by the granularity level in which they occur. Although they are all equally important, it is worth to mention just two of them, as they are supported by the majority of the automatic detection tool developed so far. They are static cycles in the dependency graph and between packages.

Static Cycles in Dependency Graph, *a.k.a.* Cyclic Dependency at class level, arises when two or more classes use each other. Cycles have effect on understandability, as the graph needs to be comprehended as a whole, maintainability, as they can have severe and unpredictable consequences, thus making it harder to change the systems affected by them, reusability, as the class in the graph can be re-used as a whole, and testability, since the classes can only be tested in their totality as a graph.

Similarly to the previous one, Cycles between packages can be created through use, inheritance, or a combination of use and inheritance. They have the same impact as the class counterpart. However, according to the authors, they are easier to solve. In most cases, this problem can be resolved through simple restructuring, for example by merging all packages participating in a cycle into one package, which will then be arranged based on better criteria.

It is worth to mention that they also put the base for automatic architectural smells detection issue: they reasonably claim that reading code is not sufficient to detect these flaws since they usually emerge not from a single class, but the interaction of many classes. So automatic detection tools are needed. They also present some tools that allow developers to understand better and analyze their code-bases, such as *JDepend*, which can analyze the dependencies between packages and classes and calculate some metrics. Hoverer, these tools are not specific for architectural smells, but their output should be the bases for architects to analyze the code-bases to spot architectural smells.

Another initial attempt to address the issue of architectural smells definition and cataloging was presented by Garcia *et al.* in 2009 [9]. They also introduce a definition of *architectural bad smell*, addressing it as "frequently recurring software designs that can have non-obvious and significant detrimental effects on system life-cycle properties, such as understandability, testability, extensibility, and re-usability". Moreover, they differentiated them from related concepts, such as architectural anti-patterns and code smells. Finally, they described a set of four representative architectural smells, namely Connector Envy, Scattered Parasitic Functionality, Ambiguous Interface, and Extraneous Adjacent Connector, encountered in the context of an industrial re-engineering [9].

This work than conveyed in a greater and more structured one, made by Garcia *et al.* themselves in 2009, which aimed to propose a preliminary example of architectural smells catalog [10]. A first extension to the definition is the explicit capture of architectural smells as design instances independent from the engineering processes that created the design. Human organizations and processes are orthogonal to the definition and impact of a specific architectural smell. In practical terms, this means that the detection and correction of architectural smells are not dependent on an understanding of the history of a software system. As a second extension to the definition, they do not differentiate between architectural smells that are part of an intended design (*e.g.*, a set of UML specifications for a system that has not yet been built) as opposed to an implemented design (*e.g.*, the implicit architecture of an executing system). Furthermore, they do not consider the non-conformance of an implemented architecture to an intended architecture, by itself, to be an architectural smell, since an implemented architecture may improve maintainability by violating its intended design. Finally, they attempt to facilitate the detection of architectural smells through specific, concrete definitions captured in terms of standard architectural building blocks — components, connectors, interfaces, and configurations. In the conclusion of this paper, they state the need for software architects to have a catalog which they can consult s to analyze the most relevant parts of an architecture without needing to deal with the intractability of analyzing the system as a whole since architectural smells tell architects when and where to refactor their architectures, as well as code smells have helped developers identify when and where source code needs to be refactored [10]. One of the main problems with this work is that the authors do not reference their detection support.

Another work, presented by Le *et al.* [11], was aimed to provide a taxonomy of architectural smells, metrics, and their impacted quality properties, relating these smells to maintenance and evolution areas as a first step toward the ultimate goal of estimating the sustainability of systems. Some of the architectural smells they propose were already touched in the ones mentioned above, *i.e.*, Ambiguous Interface, Scattered Parasitic Functionality and Dependency Cycle (*a.k.a.* Cyclic Dependency), other are new, *e.g.*, Unused Interface (which occurs when an interface of a component is linked with no other), or Duplicate Functionality (which occurs when a component shares the same functionality with other components). The significant contribution to what concerns architectural smells definition is that they provide another way to classify the smells, namely by Interface, Change, Dependency and Concern-based. Their classification considers the reason or the causes of the architectural smells rather than where they occur.

This deviation from a *locality* classification was also pointed out by Ganesh *et al.* in 2013 [12]. Their classification is also based on design principles, namely Modularity, Hierarchy, Abstraction, and Encapsulation [12]. As asserted by the authors, the rationale of this classification is that it enables an intuitive understanding of the smell, and it allows us to get a better idea of how to refactor the architectural smells. They present a proper catalog, providing for each smell its name, description, rationale, examples, violated principles, impacted quality attributes, aliases, variants, exceptions, detection strategies, and suggested refactoring. Then, they present the catalog containing all the 30 *design smells* they have classified in the four categories above.

This work was fundamental for another catalog proposal, presented by Azadi *et al.* [5]. Differently from the previous one, the criteria for the creation of this collection is particular: they propose a catalog of architectural smells for which at least one tool able to detect the smell exists, and then they outlined the main differences between detection techniques, and classification of these architectural smells according to the violation of three design principles, namely *Modularity* (the property of a system that has been decomposed into a set of cohesive and loosely coupled modules), *Hierarchy* (ranking or ordering of abstractions), and *Healty Dependency Structure*. This last principle was defined by themselves: it is aimed to group some of the already well-known best practices that involve dependencies, which are (i) the desirable acyclic nature of a subsystem's dependency structure [4], (ii) the desirable equality of stability between the two elements (classes or packages) that are involved in a dependency relationship [4] and (iii) all the principles that aim to prevent the occurrence of the Rigidity [4], which is a symptom of rotting design. They also compared their classification with the one presented by Ganesh *et al.* [12]. They decided not to consider the Abstraction and Encapsulation principles because they are more closely related to the concept of code smells. Second, the template for their catalog is mostly inspired by the one presented by Ganesh *et al.* [12], adapted to make it more suitable for architectural smell detection support. So, their template lacks ”impacted quality attributes”, ”detection strategies”, and ”suggested refactoring” sections, but has its own unique ”tools” and ”detection comparison” one.

They analyzed nine automatic tools, five commercial and four open-source (which will be described in the following section), and they got a collection of 12 architectural smells. Some of them have already been described, such as Cyclic Dependency, Scattered Functionality, Ambiguous Interface, and Unused Interface, but others were not mentioned before. Two of them are the most relevant for this thesis and were described more deeply: Hub-Like Dependency and Unstable Dependency.

Hub-Like Dependency Hub-Like Dependency, *a.k.a.* Hub-like Modularization or Link Overload, is a smell that arises when an abstraction or a concrete class has (outgoing and ingoing) dependencies with a large number of other abstractions or concrete classes. It violates both the *modularity* and *healty dependency structure* [5]. The former is violated since classes affected by this smell are usually overloaded with responsibilities, highlighting a severe coupling problem between the hub-like class and all the other classes involved. The latter is violated since the close coupling described above implies a dependency structure, particularly unhealthy [5]. That is because a change in a "hub" class will require to adapt at least all the dependent classes. Furthermore, the classes on which the "hub" class depends can be affected by the changes in the "hub" class and their change will affect the "hub" class. Therefore even the slight adjustment in a system affected by this smell will generate a ripple effect in all the involved classes.

Unstable Dependency Unstable Dependency, *a.k.a.* Unstable Interface, describes a subsystem (component) that depends on other subsystems that are less stable than itself, and because of this dependency the more stable files (classes) are changed frequently with the other files. It violates the *healthy dependency structure* [5], since the dependencies that allow this smell to arise increment significantly the change-proneness of the more stable subsystem and they also directly violate the *Stable Dependencies Principle* [4].

Although these are not the solely they classify in their work, along with Cyclic Dependency, they constitute the main object of study in this thesis, and so deserved a special mention.

2.2 Automatic Architectural Smell Detection

As said in the previous section, Azadi *et al.* have constructed their catalog, considering only the architectural smells detected by at least one tool. In this section, some of these tools are further discusses, presenting them alongside their benefits and drawbacks.

Designite [7] is a standalone commercial tool for automatic *design* and *code* smell detection, able to spot 19 design and 11 code smells (among them Cyclic Dependency, Hub-Like Dependency, Unstable Dependency) in C# code-bases. It was developed to overcome some issues in automatic smell detection, *i.e.*, lack of automatic detection for some smells, lack of support for other programming languages except Java, lack of support in detection parameters and thresholds. The tool uses NRefactory to parse C# code and prepares the Abstract Syntax Tree (AST). The source model layer accesses the AST and prepares a simple hierarchical meta-model. The meta-model captures the required source code information used by the back-end of Designite to infer smells and compute metrics. Besides detecting smells, Designite is also able to compute 30 object-oriented metrics, and to analyze multiple version of the same project to perform trend analysis. More recently, it was also developed an open-source version able to support detection in Java code-bases.

Mo *et al.* presented another interesting tool., named Hotspot Detector [13]. It can detect five *hotspot patterns*, as defined by the authors (among them Cyclic Dependency). The tool exploits the *Design Structure Matrix* (DSM). A DSM is a square matrix, whose rows and columns are labeled with the files contained in the project. If a cell in row x, column y, c:(rx,cy), is marked, it means that file x is structurally related to file y or evolutionarily related. The cells along the diagonal represent self-dependencies. The tool takes input: (i) the DSM file for structural dependencies, A DSM file containing the evolutionary coupling information, and a clustering file containing the clustering information of the files. Given these inputs, the Hotspot detector will output a summary of all the architecture issues and the files involved in each issue. However, the tool is not currently available.

An interesting commercial tool is AI Reviewer [14], a code analysis and measurement tool, capable of performing fully automated code reviews of large C++ projects, based on the well known SOLID [15] principles of object-oriented design. It cannot only perform some standard code smells analysis, *e.g.*, God

Class, Refused Bequest, Feature Envy, but also to detect violations of the SOLID design principles and compute and report dozens of code metrics. AI Reviewer combines static code analysis with heuristic techniques to perform its investigation and establish its findings. To what concerns the architectural smells, it can detect Cyclic Dependency and Unstable Dependency.

Last but not least we have ARCAN [6], presented by Fontana *et al.* in 2017. This tool, written in Java, can spot three architectural smells, namely Cyclic Dependency, Hub-Like Dependency, and Unstable Dependency. The detection techniques inside Arcan exploit graph database technology, allowing for high scalability in smells detection and better management of large amounts of dependencies of multiple kinds. In particular, it relies on the library Apache Tinkerpop to interface with graph databases, to allow the exploitation of different graph databases backends. Thus ARCAN can exploit both a local database instance or a remote Neo4j instance. Cyclic Dependency are detected establishing rules to detect particular cyclic shapes, *i.e.*, tiny cycles, cliques, simple, and many others. Hub-Like Dependency are detected in classes figuring out the conditions where these false-positive instances could arise. So the detector excludes from the dependencies all the external relationship to other libraries, *i.e.*, `java.util.*`. Classes of this form are rather simple because they most likely use default functionalities (e.g., lists). Conversely, classes that are frequently used and implement the system's main functionalities exhibit the opposite pattern. Finally, Unstable Dependency, detected on packages, is detected computing Martin's Instability metric [4]. They defined a filter to remove false positive instances. Since a package is considered affected by this smell only if it depends on another package less stable than itself, to examine *only* the dependencies which cause the smell, they defined the *Degree of Unstable Dependency*, that is the ration between *Bad Dependencies* over the total number of dependencies, considering *Bad Dependencies* a dependency that points to a less stable package. The reason for this is that they conjectured that a package with a small number of bad dependencies might not be a smell, and this formula helps to filter misleading results. A package considered smell presents a value of *Degree of Unstable Dependency* higher than 30%. On the one hand, this may remove some false positive instances; however, this implies a heuristic and a threshold formulation that need to be validated. This validation process was conducted in a preliminary stage, applying it in a case study made up of two projects. The evaluation was carried out by direct observation of Arcan results. As a result, the tool scored a 100% precision on both projects and 66% and 60% recall. However, at the moment, ARCAN is not openly available.

2.3 Empirical Studies on Architectural Smells

In this section, some of the few empirical studies that can be related to architectural smells are explored. The first explored one was conducted by Le *et al.* [11], the one that results in a preliminary form of an architectural smell taxonomy. The goal of this work was to relate these smells to maintenance and evolution areas as a first step toward the ultimate goal of estimating the sustainability of systems. They detected architectural smells and quality metrics, in particular architectural decay metrics. Then they developed a preliminary taxonomy relating the metrics that can be used to measure the architectural smells. To verify our proposed taxonomy, they conducted pilot studies where they used information about the bugs found in these systems as an indicator of the systems' sustainability. Then, they analyzed the subject systems to find their architectural smells and computed values of architectural decay metrics. After that, they studied the correlations between decay indicators and numbers of bugs in each software system, intending to discover which proposed architectural decay indicators can be used to estimate system sustainability accurately, applying the Pearson correlation coefficient. Their preliminary findings from nine subject systems analyzed show strong correlations between all three architectural decay indicators and the number of bugs, for some of the systems, but not for others. Although they have not discovered the reasons for this correlation yet, they have proposed further investigating correlations between architectural smells with characteristics of bugs.

Nevertheless, this pilot study shows some promising results using architecture smells to predict system sustainability.

Another study, conducted by Macia *et al.* [16], presented an exploratory analysis that investigated to what extent the automatically-detected code anomalies are related to problems that occur with an evolving system's architecture. They analyzed code anomaly occurrences in 38 versions of 5 applications using existing detection strategies. The outcome of their evaluation suggests that many of the code anomalies detected by the employed strategies were not related to architectural problems. Even worse, over 50% of the anomalies not observed by the employed techniques (false negatives) were found to be correlated with architectural problems.

The last one presented, related to architectural smells, is a case study conducted by Martini *et al.* in a large software company [17]. In this paper, they conducted a multiple case-study on several architectural smells detected in four industrial projects. They conducted an in-depth investigation with a questionnaire, interviews, and inspection of the code with the practitioners. They evaluated the negative impact of the technical debt detected by the architectural smells, their difficulty to be refactored and the usefulness of the detection tool. The results show that practitioners appreciated the help of automatic detection and that they prioritize refactoring architectural debt that causes more negative impact despite the higher refactoring effort.

These are not the solely related work to architectural issues; however, they are not are mainly focused on architectural smells matter and therefore are not listed.

Chapter 3

Brunelleschi: A Novel Tool

In the previous chapter, the efforts made so far to tackle architectural smells have been presented. However, there are both some theoretical and practical limitations. The most important theoretical limitation is that, some smells still lack a clear-cut definition, and thus some boundary cases are not properly managed. Furthermore, smell instances could not easily be classified in one definition or another. This is still an open challenge, that requires a better understanding of architectural smells. About the practical limitations, most of the presented tools are either unavailable or distributed under a commercial license (except for Designite, whose team has recently released a free standalone version for Java). This strongly limits research activities, as architectural smells keep being uncovered without a proper tools able to spot them. So, to this aim, Brunelleschi was conceived: a free open-source tool able to detect three types of architectural smells in Java projects, *i.e.*, Cyclic Dependency (both at class and package level), Hub-Like Dependency (at class level), and Unstable Dependency (at package level). The following sections provide a comprehensive description of Brunelleschi: Section 3.1 proposes a general overview of the tool; Section 3.2 documents the architectural choices made and implemented; In Section 4 the tool validation procedure is documented and finally, in Section 6.2.1 a proposal of future enhancements is discussed.

3.1 Overview

Brunelleschi, is an automatic architectural smell identification tool for Java projects, integrated and deployed directly in IntelliJ IDEA, able to detect 3 architectural smells in source code, namely Cyclic Dependency, Hub-Like Dependency, Unstable Dependency. For each of these architectural smells a detection rule was constructed, deriving it directly from the smell definition [5], and then implemented in Brunelleschi. This implementation exploits the dependencies present among software components, which are represented through a dependency graph (*i.e.*, a direct graph whose nodes represent the software components and edges the dependencies among them). To be more precise, Brunelleschi analyzes source code using IntelliJ IDEA Psi library, and builds two graphs, one for class-level dependencies and one for package level dependencies. After that, the detection rules are run on these representations in order to spot architectural smell candidates. The smells' definitions along with their detection rules are defined in the following.

Cyclic Dependency Cyclic Dependency is a smell which arises when two or more architectural components depends on each other directly or indirectly [5]. This means that in a directed graph representation of their dependency they form a cycle. Cyclic Dependency is detected applying Gabow's algorithm presented in Path-based depth-first search for strong and biconnected components by Gabow [18], and it is run on both class and package dependency graph. Since in a directed graph a strongly connected component requires that the nodes are linked thorough at least one cycle, finding all the strongly connected components allows Brunelleschi to find every cycle present in the graph as well.

Hub-Like Dependency Hub-Like Dependency occurs when an abstraction or a concrete class has (outgoing and ingoing) dependencies with a large number of other abstractions or concrete classes [5]. As can be seen, this is one of those definitions that are not clear-cut, as large can mean any number. Hub-Like De-

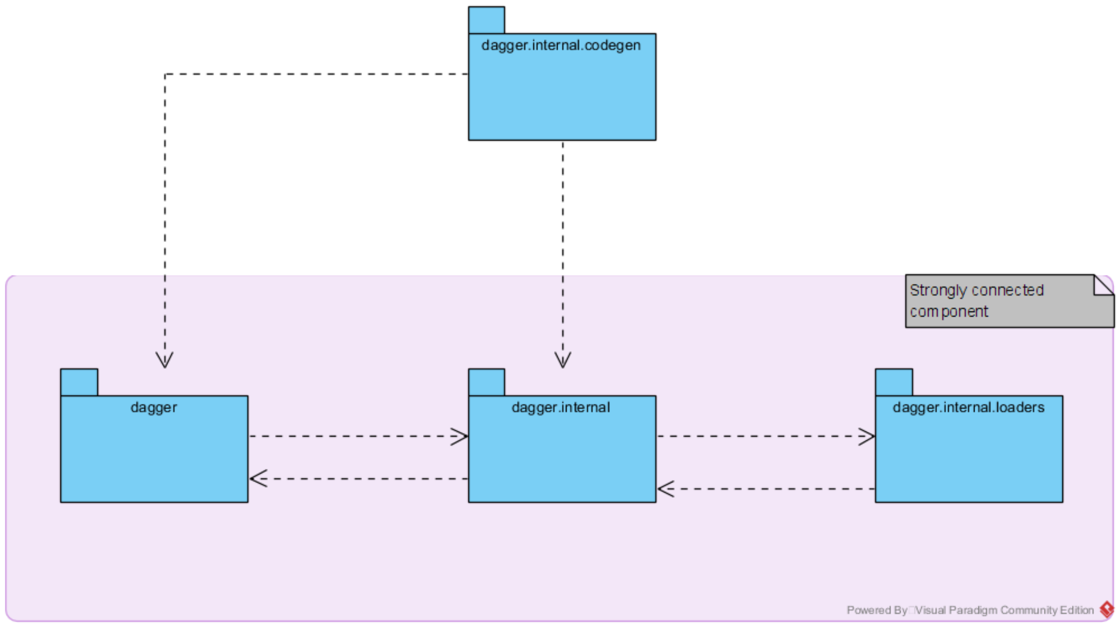

dagger.internal.codegen
Strongly connected component
dagger
dagger.internal
dagger.internal.loaders
Powered By Visual Paradigm Community Edition

Figure 3.1: Example strongly connected component for Cyclic Dependency detection

pendency instances are spotted considering the number of incident edges in a dependency graph vertex: if a node shows up at least 20 ingoing and outgoing edges, it is marked as smelly. The choice of the threshold (20) is not left to choice. This is the rule applied by DESIGNITE [7], and the one that is also reported in the catalog [5], which has proved itself to be valid. However, it is worth saying that when building the graph, the tool does not take into account dependencies from external libraries, *e.g.*, Java standard library, but only those parsed by INTELLIJ IDEA, specifically the ones present in the main source folder. This means that the potential false positive problem that, for instance, ARCAN [6] faced, does not subsists.

Unstable Dependency Unstable Dependency describes a component that depends on other components that are less stable than itself. This definition directly comes from Martin's Instability metric [4], and so a direct calculation of that is exploited to spot it in the code-base. For each package in the graph, the metric is computed, employing the number of ingoing and outgoing edge count, as shown in Figure 3.2. Then, the instability of each successor node (namely a node reached

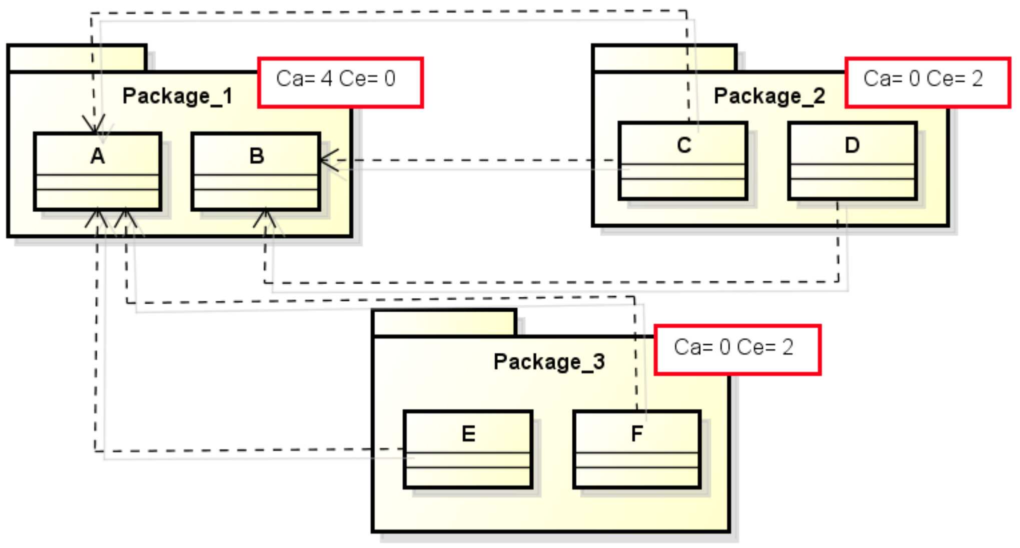


Figure 3.2: Example of Instability Metric calculation

by an outgoing edge, *i.e.*, a package the considered package depends on) is taken into account. If at least one successor is more unstable than the considered node, then the node, and so the package, is labeled as an Unstable Dependency instance.

Once performed these analyses, Brunelleschi gives as output a CSV file for each detected smell, containing the project name, the qualified name of the affected components, the component type (class or package), and the name of the smell. Besides, it generates a summary CSV file, containing the total number of the detected smells, and the count of each type of smell detected. This kind of output subsists since the tool is still in an experimental phase, and so this output format eases the results' analysis. As discusses in Section 6.2.1, a better output is one of the proposed enhancements.

Brunelleschi's usage is currently pretty straightforward, so it does not need a completely separated section to explain it. At the moment there is only one way to run Brunelleschi, and that is using the menu button present on the IDE main toolbar. Once launched, a progress bar shows up, making the user understand that the tool is performing code analysis, and eventually, a pop-up with a success message appears, to notify that the operation is concluded. Results can be found in a specific directory, which at the moment is USER_HOME/.brunleleschi/*project_name*. As said before, a better and more configurable output will be object of future enhancements.

3.2 Architecture

In the following section, BRUNELLESCHI's architecture is documented. Before diving into this, it is worth spending some words about the used notation: according to Ian Sommerville [19], UML is not employed to document the tool dependency structure, but rather it is preferred an informal "box-and-arrow" notation, to simplify the reading and to better convey the intended information, as "Uncle Bob" [4] also did in his book.

The first design decision to report is that BRUNELLESCHI is an INTELLIJ IDEA plug-in and so it is deployed, and executed, directly into it. Moreover, it is shipped through GitHub and (soon) through Jetbrain's official plug-in repository. Despite this, and although it may seem counterintuitive, it could be stated that INTELLIJ IDEA is the real *plug-in* for BRUNELLESCHI, and not the opposite, employing the word "plug-in" as Martin intended [4]. This is necessary, since the tool was developed following the book's guidelines, making its architecture, using a popular buzzword, an *Onion*, or *Clean*, architecture.

If this might seem a bit confusing, Figure 3.3 clearly depicts the scenario, in which a high-level picture of BRUNELLESCHI's main components is taken. Before diving into its description, some other words must be spent about the notation. Intuitively, boxes represent components, and arrows the dependencies among them. The double straight lines, on the other hand, indicate an *architectural boundary*, as intended by Martin: a separation between *high-level* components and *low-level* ones. Boundaries are fundamental to define that dependency rule [4] which makes an architecture *clean*, and which was followed in the tool development: dependencies across the boundaries can be traversed only in one direction. This prevents high-level components, implementing application's business rules, to depend on

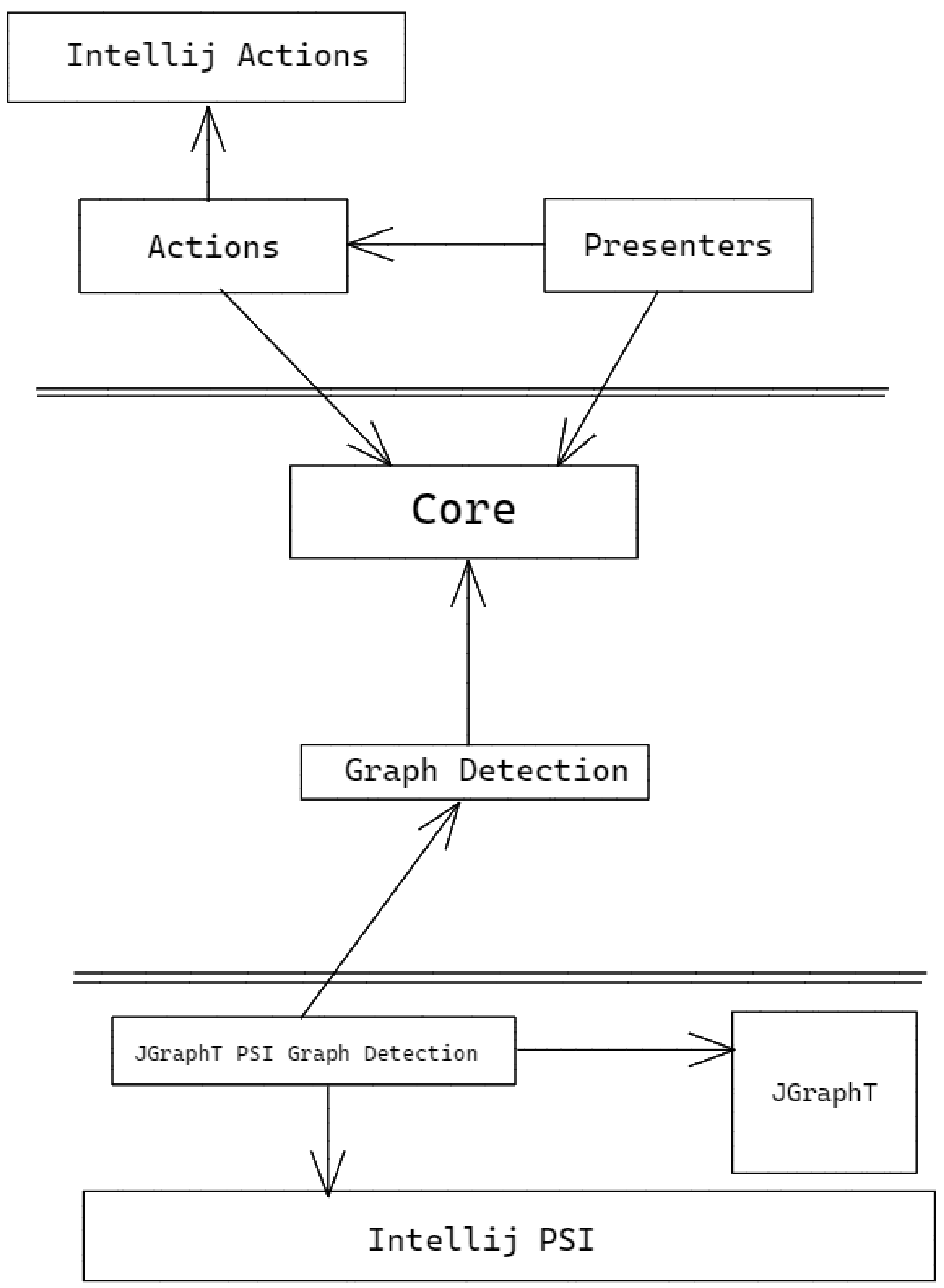


Figure 3.3: Representation of Brunelleschi's High-Level Components

low-level details, such as GUI or Databases or I/O devices. To make this happen, the *Dependency Inversion Principle* and *Open/Closed Principle* [15], which are not discussed here, are largely used, *e.g.*, the JGraphTPSIGraphDetection component is a valid representative of that. If this keeps to be obscure, a description of each showed component is provided in this section, specifying the design decision made and clarifying all the *plug-in matter*.

Core As the name could easily suggest, Core is the fundamental component, placed in the innermost circle of the onion architecture. It contains the 3 entity classes (ArchitecturalSmell, Component SmellDetector) which implements the basic business rules, and the Interactor classes, implementing the use cases, that currently is one: FindSmellInteractor. The SmellDetector class deserve a special mention, because it is implemented through a decorator [20], in order to combine at runtime different kinds of analyses. To construct it, the builder pattern [20] is used, so to keep the clients unaware of the concrete implementations.

Graph Detection This components contains the concrete implementation of the detection rules described above, so that we have a class for each rule, that are, of course, the concrete decorators. As said, these rules exploit computations made on a direct-graph representation of the project under analysis. Thus, in this component, an abstract DependencyGraph class is defined, which provides the abstract methods required by the detectors to perform their analysis. In this scenario, whenever a new detection strategy needs to be implemented, no line of code must be changed, but a completely new component can be written, providing classes implementing the abstract smell detector defined in the Core component. As said, graph representation is also an abstraction, and a factory class is provided.

JGraphTPSIGraphDetection This is one of the components that can be classified in the *low-level details* category, despite being a fundamental component for this high-level description. This component contains a dependency graph concrete implementation, which relies on a graph library, namely JGraphT (as the figure shows). It also contains the concrete implementation of the DependencyGraphFactory, which rely on Psi and its algorithm to construct a dependency graph, both for classes and packages. Again, if a new graph library, or a different algorithm for dependency class construction (either using Psi or not) needs to be used, a new component could be written, and no code needs to be changed in this one.

Presenter On the other side of another architectural boundary there is the Presenter component. As the name suggests, this component contains classes which are responsible to present output to users. Classes in this component transform the results obtained from Interactor classes into a more convenient form for end user: this is the case of the CSVPresenter class, which is responsible to produce the aforementioned CSV files that represent the current output. Presenters are all child of an AbstractPresenter, so that, whenever a new output strategy is needed, it only requires to write another concrete presenter.

Actions This is the most *unstable*, and the most low-level component, the one that makes Brunelleschi callable by IntelliJ IDEA. Classes in this component play the role of controllers. Here also resides configuration classes, that wire up all the concrete dependencies, providing them to actions and making them actually work. Here, an instance of command pattern is found, used in order to not make the Interactor change, whenever a different type of analysis needs to be performed. Commands assemble the concrete detectors, using the builder, and then are executed by the Interactor class. At the moment, only a single command is implemented, FindAllSmellCommand, but it worth nothing to say that whenever a new type of analysis needs to be performed, it just need to write a new command that wires up the detectors in a different way.

It is easy now to understand why Brunelleschi is not fully dependent by IntelliJ IDEA, and rather, the IDE is a *plugin* for it: since the core components do not depend on anything, they can be easily extracted in a completely separated software. Since the IDE's libraries can easily be substituted, they become a plug-in to the core functionalities. And so, substituting the components which were *kindly offered* by the IDE, the tool could properly work again. In the end, the tool resides in the IDE only for convenience, as the parser and other code-related libraries are easily reusable. Making the tool's evolution open to any kind of future changes, including a *divorce* from IntelliJ IDEA, is clearly the most important design decision made.

Chapter 4

Brunelleschi's Validation

In this chapter, the validation of the tool is discussed. The *goal* of the study is to evaluate BRUNELLESCHI with the *purpose* to asses its precision in detecting architectural smells. Specifically, the research question that the study aims to answer is the following:

RQ$_0$. *What is* BRUNELLESCHI*'s precision in detecting architectural smells?*

The context of the study is made up by 36 Java open source projects taken from GITHUB, whose characteristic are resumed in table 4.1. Recall is temporally neglected, as will be further explored in section 4.2, since the tool is rule-based and not heuristic-bases, and so, because of time constraints, in this phase there is no need to validate it. However, as following discussed, a further empirical study is planned to asses both precision and recall on a larger dataset.

4.1 Methodology

The methodology applied for the analysis is pretty straightforward: for each project, the detection was run and then the results collected using the CVS format explained above. The tool reported a total amount of 1989 architectural smells (385 Cyclic Dependency, 381 Hub-Like Dependency, 1223 Unstable Dependency).

#	Name	KLOC.	Domain
1	AndroidUtilCode	51.4	Android Library
2	BaseRecyclerViewAdapterHelper	4.6	Android Library
3	Butterknife	13.7	Android Library
4	Hystrix	63.5	Library
5	OpenRefine	100.9	Tool
6	MPAndroidChart	31.6	Android Library
7	Tinker	45.1	Android Library
8	Fastjson	185.1	Library
9	Apache Dubbo	221.1	Framework
10	Glide	84	Android Library
11	PhotoView	1.8	Android Library
12	Fresco	107.8	Android Library
13	Rebound	5.1	Library
14	ExoPlayer	200.6	Android Library
15	Guava	660	Library
16	Guice	82.8	Framework
17	j2objc	100	Tool
18	EventBus	6.8	Library
19	GreenDao	23.4	Android Library
20	Jenkins	60.6	Tool
21	JUnit4	37	Library
22	Mockito	67.6	Library
23	MyBatis-3	82.4	Library
24	Netty	366.6	Framework
25	Logger	1	Android Library
26	Realm Java	110.5	Android Library
27	Scribe Java	15.4	Library
28	Jadx	71.5	Tool
29	Socket.IO client	3.7	Library
30	Dagger	11.5	Android Library
31	Java Poet	10.5	Library
32	Leaky Canary	5	Android Library
33	Moshi	18.1	Library
34	OkHttp	38.1	Library
35	Picasso	7.7	Android Library
36	Retrofit	31.1	Library
37	Zxing	56.1	Android Library

Table 4.1: Basic characteristics of the software project considered

These results were manually validated. However, since it was time and resource consuming to evaluate all the 1989 instances, from these results, a statistically stratified significant sample was taken, which underwent to a manual validation analysis. So, having a confidence level of 95% and a confidence interval of 5, the resulting sample counted 323 architectural smells, which keeping the proportions among them is made up of 61 Cyclic Dependency, 62 Hub-Like Dependency and 200 Unstable Dependency.

The Cyclic Dependency instances validation relied on the Dependency Structure Matrix (DSM) provided by INTELLIJ IDEA itself [21]. DSM a method that helps developers to visualize dependencies between the parts of a project (modules, classes, and so on) and highlights the information flow within the project.

As shown in Figure 4.1, components are displayed in a matrix form, and for each cell, there is a number displaying the number of component usages. On the matrix, all dependencies always flow from green to yellow: when selecting a row, green annotations show dependent components, while the yellow ones show components on which the selected components depends. Following this flow, it was possible to reconstruct the portion of the dependency graph that BRUNELLESCHI reported as a Cyclic Dependency, thus verifying if it was an actual cycle.

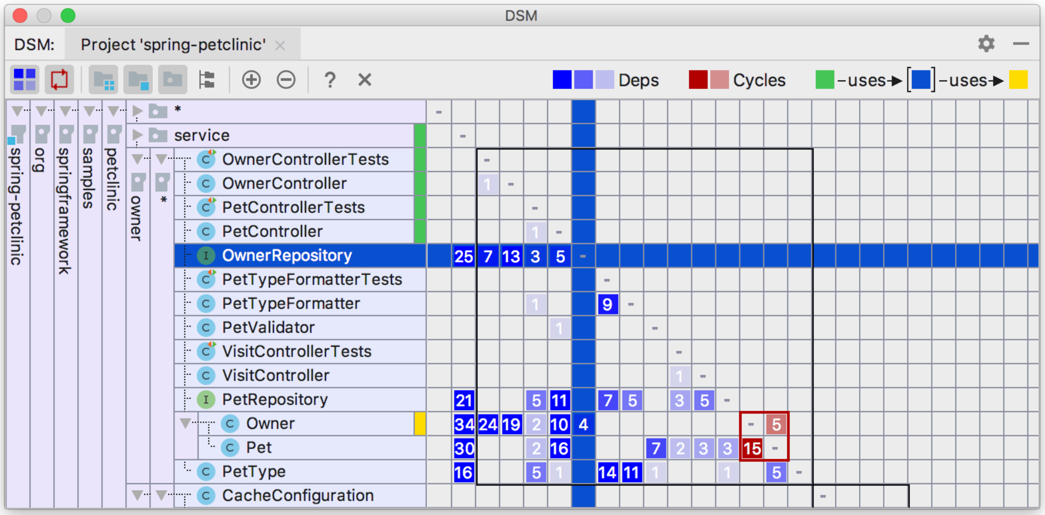
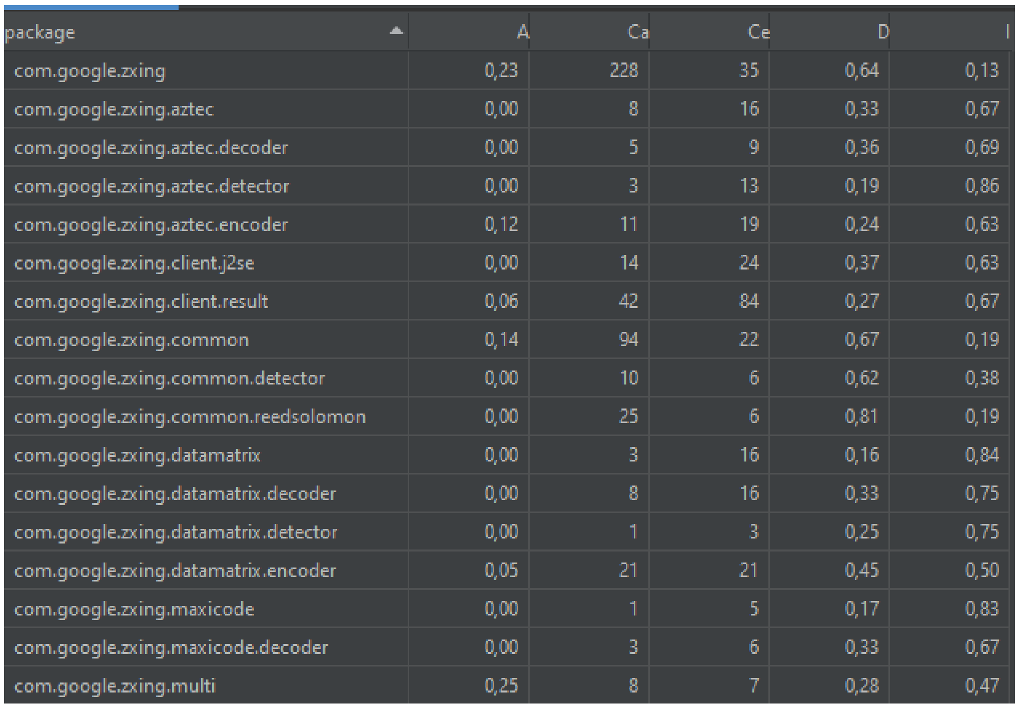


Figure 4.1: An example of DSM tool window



package	A	Ca	Ce	D	I
com.google.zxing	0,23	228	35	0,64	0,13
com.google.zxing.aztec	0,00	8	16	0,33	0,67
com.google.zxing.aztec.decoder	0,00	5	9	0,36	0,69
com.google.zxing.aztec.detector	0,00	3	13	0,19	0,86
com.google.zxing.aztec.encoder	0,12	11	19	0,24	0,63
com.google.zxing.client.j2se	0,00	14	24	0,37	0,63
com.google.zxing.client.result	0,06	42	84	0,27	0,67
com.google.zxing.common	0,14	94	22	0,67	0,19
com.google.zxing.common.detector	0,00	10	6	0,62	0,38
com.google.zxing.common.reedsolomon	0,00	25	6	0,81	0,19
com.google.zxing.datamatrix	0,00	3	16	0,16	0,84
com.google.zxing.datamatrix.decoder	0,00	8	16	0,33	0,75
com.google.zxing.datamatrix.detector	0,00	1	3	0,25	0,75
com.google.zxing.datamatrix.encoder	0,05	21	21	0,45	0,50
com.google.zxing.maxicode	0,00	1	5	0,17	0,83
com.google.zxing.maxicode.decoder	0,00	3	6	0,33	0,67
com.google.zxing.multi	0,25	8	7	0,28	0,47

Figure 4.2: An example of Martin's metric calculation

Hub-Like Dependency validation relied on DSM as well: counting the number of non empty cells on a component row allowed the verification of the detection rule (*i.e.*, the number of total dependencies higher than 40). Finally, to validate Unstable Dependency an IntelliJ IDEA plugin was employed, named Metrics Reloaded, which is able to compute Martin's metrics [4, 15], alongside with DSM. As for Cyclic Dependency, exploiting the dependency flow, it was possible to reconstruct the efferent and afferent edges, and thus manually compute the instability metric. Then, applying the same approach, it was possible to calculate the instability metric for the components which the candidate depends on. Thus, it was easy to evaluate if a candidate actually depends on a more unstable component.

4.1.1 Results

Table 4.2 shows the obtained results, both global and for each specific smell. It worth noting that only for Cyclic Dependency a precision score lower than

	TP	FP	Precision
Cyclic Dependency	41	20	60%
Hub-Like Dependency.	62	0	100%
Unstable Dependency	200	0	100%
Total	303	20	93%

Table 4.2: BRUNELLESCHI *precision scores*

100 was calculated. So, a further investigation was conducted and it showed that only class-level instances showed the presence of false positive, while for package-level no false positive instance was found. So, further analyzing these false positive instances, it was found that only cycles that presented annotations, abstract classes and interfaces presented false positive instances. Thus, it may be concluded that the problem could be addressed in the graph building phase. Since BRUNELLESCHI exploited the reference search provided by Psi API, when edges between classes are defined, some of them are not real dependencies. Although it is still not clear why some *false* edges may appear, this is might caused by the fact that annotations and interfaces are considered as normal classes by INTELLIJ IDEA. For instance, in the Dagger project, there is an annotation Module, located in the dagger, that is labeled as a member of a Cyclic Dependency alongside with another class, ObjectGraph, belonging to the same package. However, performing manual validation with DSM, the Module annotation do not present any outgoing dependency, which is reasonable, since it is an annotation. However, since the error is limited, and there is only a lack of precision, it could not be considered as critical. Despite this, a future corrective enhancement has been already planned, as the current graph building algorithm is inefficient both in time and space. This is an further reason to either fixing or replacing it.

Finding 1*.* BRUNELLESCHI *scored a 93% precision.*

4.2 Discussion on Recall

BRUNELLESCHI has proved itself to have a good precision score. However, recall was neither considered nor evaluated. This was a deliberate decision. It must be considered that the detection rules mentioned before are not heuristics, but they were derived directly from the smells' definitions [5]. Since they can be considered specifications, and not heuristics, there is no need of recall validation, but rather a testing activity, to be *reasonably sure* that they were correctly implemented. Considering the case of an actual smell not detected by BRUNELLESCHI, there must be only one possible reason to explain it, *i.e.*, that the rule was badly implemented. Since in the testing phase, all the rules were considered, there could be only two possible causes: either there is an unspotted bug, or the recall is maximum. However, this is the classical testing problem, and nobody can be 100% sure that a software is bug-free. So, as far as it is known, the only possible scenario is the one with maximum precision.

Chapter 5

Empirical Studies

5.1 Empirical Studies Design

This chapter focuses on the design of the three empirical studies that were carried out to widen the knowledge about architectural smells. Since BRUNELLESCHI has proven itself to have good performances, it can be exploited to carry out quantitative studies. The *goal* of these studies was to investigate architectural smells with the *purpose* of better understanding their diffusion, the relationships among them, whether developers are aware of their existence in code-bases, and whether they need automatic detection tools. The perspective is of both researchers and practitioners, who are interested in discovering the potential issues caused by architectural smells. To achieve this goal, four research questions are explored:

RQ$_1$. *To what extent are architectural smells diffused in software projects?*

RQ$_2$. *Does an architectural smell instance imply the presence of other smells?*

RQ$_3$. *To what extent developers are aware of architectural smells and their impact in code-bases?*

RQ$_4$. *Do developers need an automatic tool for architectural smell detection?*

Section 5.1.1 describes how the former two RQs were addressed, while Section 5.1.2 contains the design of the qualitative study addressing the latter two RQs.

5.1.1 The quantitative studies

The *context* of the studies is made up of (1) projects and (2) architectural smells. These were used to address the first two research questions mentioned before. In the following, a description of each of them is provided.

Projects For these experiments was used a dataset of 36 Java Open-Source projects available on GitHub, that were already used for Brunelleschi's validation and whose characteristics are resumed in Table 4.1. The choice of the projects to consider was not random, but based on convenience. Indeed, an initial set of 50 projects was considered, based on their popularity and reported in [22]. The number of analyzed projects was then reduced to 36, as at the time of the analyses, their builds were not passing and so it was not possible to compile them, and run Brunelleschi. Two things are immediately noticeable: the first is that most of the projects are Libraries, and the second is that there is a great variance in the number of KLOC. While the latter is quite understandable, as this conveys a higher generalization of the experiments' results, the former needs some explanations: it is quite expected to find that the majority of the considered projects were libraries, because of popularity (stars on GitHub). However, even software systems were found, although being tools in most cases. For the same reason, also some frameworks were taken into account.

Architectural Smells The choice of the architectural smells to investigate, of course, was based on convenience as well. The three smell selected, Cyclic Dependency, Hub-Like Dependency, and Unstable Dependency, are the one that could be detected by Brunelleschi, and so the only one whose measurement was accessible at that moment. Despite the convenience matter, the tool also well performed in the validation study, discussed in Chapter 3, showing a precision score of 93% and so it can be considered fairly reliable. In addition, these smells are the ones with a clear-cut definition [5], and thus the one that could more easily investigated in code-bases. Moreover, since their simplicity, they are good candidates for a qualitative analysis among developers.

Data Analysis and Collection

To address the two research questions, a three step data analysis was performed:

1. Identification of the architectural smells;
2. Descriptive statistic analysis;
3. Co-occurrence analysis.

The first step is pretty straightforward: BRUNELLESCHI was run on the projects, and the results were gathered. Then, the summary files of the different projects were aggregated in a single file, containing the smell instances detected for all the projects. After that, the gathered data were analyzed using basic statistic instruments (mean, median, box-plots, *etc.*) exploiting the tool R. It is worth to say that not only the absolute number of smells in a software project was analyzed, but also the number of smells instances weighted by package and class number. Finally, the third step is aimed to asses how often the presence of architectural smells of a given type (*e.g.*, a Cyclic Dependency) implies the presence of another smell of a different type (*e.g.*, a Hub-Like Dependency). Specifically, for each smell type as_i, the percentage of co-occurrences in a class/package with of another architectural smell type as_j was computed. Formally, for each pair of architectural smells (as_i, as_j), the percentage of co-occurrences were computed with the following formula:

$$\textit{co-occurrences}_{as_{i,j}} = \frac{|as_i \wedge as_j|}{|as_i|}, \text{with } i \neq j$$

where $|as_i \wedge as_j|$ is the number of co-occurrences of as_i and as_j and $|as_i|$ is the number of occurrences of as_i. It must be noted that $\textit{co-occurrences}_{as_{i,j}}$ differs from $\textit{co-occurrences}_{as_{j,i}}$, as the formula's denominator changes from $|as_i|$to $|as_j|$.

Survey Questions	Type
Section 1	
What is your definition of 'software architectural problem'?	Free Text
For each of the mentioned smells	
To what extent does this definition allow you to understand the kind of problem that it's describing?	Point Likert Scale (from poorly understandable to very understandable)
According to you does the definition provide a complete picture of the problem?	Single Choice
If not may you please elaborate on the answer by providing us with more details on why you do not believe the definition is complete?	Free Text
Section 2	
For each of the mentioned smell	
In your opinion to what extent does the presence of a -smell name- problem impact the maintainability of a software system?	Point Likert Scale (from low impact to high impact)
If you assigned 3 to 5 to the previous question may you please elaborate on the answer by providing us with more details on the impact of a -smell name- on maintainability?	Free Text
In your opinion to what extent does the presence of a -smell name- problem impact the understandability of a software system?	Point Likert Scale (from low impact to high impact)
If you assigned 3 to 5 to the previous question may you please elaborate on the answer by providing us with more details on the impact of a ¡smell name¿ on understandability?	Free Text
According to your experience do you think that a particular organization of the development community (e.g. a community following a formal communication strategy among team members) could influence the arising of architectural problems?	Single Choice
If yes may you please elaborate on the answer by providing us with more details?	Free Text
According to your experience do you think that there are additional situations/circumstances of the development process that may lead to the emergence of architectural issues?	Free Text
According to your experience do you think that architectural issues may cause additional problems for either the development community or the source code being developed?	Free Text
Section 3	
What is your gender?	Single Choiche
What is the highest degree or level of school you have completed?	Single Choice
What is your current employment status?	Single Choice
Your experience as a developer (years):	Free Text
What is your current role in your company/organization?	Free Text
What is the size of the team you currently work with?	Free Text

Table 5.1: Questions presented on survey

5.1.2 The qualitative study

The last two research questions were addressed through a qualitative study, conducted as a large-scale survey. The survey is made up of three sections, and the complete set of questions can be found in table 5.1. The goal was to assess the potential need and utility of an automatic architectural smell detection tool, indirectly: asking developers how aware they are of architectural problems and the extent to which they could affect their code-bases. Then, the most crucial question is asked directly, of course: whether an automatic tool could help them.

The first section is designed to investigate how much the definitions of the three considered smells are clear and complete. They are asked to rate the understandability of those definitions using a score from 1 (poorly understandable) to 5 (high understandable) and possibly provide further details about the reason they rated as poorly understood.

The second section is about the impact and the emergence of architectural problems. For each of the considered smells, the participants were asked to rate the impact that the considered smell could have on the code-base *understandability* and *maintainability*, choosing a score in a range from 1 (low impact) to 5(high impact). They were then asked to provide some opinions regarding other scenarios in which these problems could arise and if the structure of their development community could influence it.

In the third section, some *personal* information is asked, such as the participant's gender, degree, current working position, and the size of their current working team.

The survey was constructed using Google Form service and then spread to the developers' community through social networks and personal acquaintances. Both students and practitioners were taken into account in order to have a broader spectrum of knowledge. To this aim, the personal programming experience was considered as a potential co-factor in the analysis of the results.

5.2 Results

In this section, a discussion about the results of the experiments is presented and their validity. Section 5.2.1 discusses the results of the quantitative studies, while section 5.2.2 discusses the qualitative study outcomes.

5.2.1 Quantitative Studies Outcomes

In this section the findings for $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ are presented and discussed. Table 5.2 shows the raw data coming from Brunelleschi analysis on the considered projects. The table shows the total amount of 1989 architectural smells found. What already emerges from this table is that no project is entirely smell-free, except for a few *noble* ones: only 3 out of 36 projects show no instances of any of the considered smell. This means that architectural smells are a fairly diffused problem.

Looking at Table 5.3 it is possible to note the distribution of the architectural smells in the code-bases. As noticeable in the first table, the number of smells found in a single project ranges from 0 is 326. In this table, it is also possible to see the mean number of smells per project, which is 55. This is a reasonably large number that indeed confirms the thesis that architectural smells are a common and unnoticed problem. However, this value is absolute, not considering the differences between large and small projects. For this reason, the weighted mean by package and classes was computed as well, considering the different levels of granularity of the smells instances. It is evident that the means are slightly increased, signifying that the problem is even more diffused than expected. Looking at both of the tables, appears that Unstable Dependency is the most diffused smell, with a 61% of occurrences over the total. This also emerges from the pie diagram presented in Figure 5.1.

This outcome was further investigated in order to understand why this happens. One probable explanation could be addressed by recalling both Martin's definition of Instability and Dependency Rule [4]. An *unstable* component is a

Name	Packages	Classes	Cyclic D.	Hub-Like D.	Unstable D.	Total Smells
AndroidUtilCode	60	273	6	1	16	23
BaseRecyclerViewAdapterHelper	32	93	2	0	3	5
Butterknife	11	156	2	1	1	4
Dagger	19	123	9	0	4	13
Dubbo	442	2244	63	25	195	283
EventBus	10	96	8	1	4	13
ExoPlayer	82	981	0	0	0	0
Fastjson	243	2902	11	13	28	52
Fresco	184	1031	40	18	80	138
Glide	62	670	11	10	61	82
GreenDao	36	245	16	7	12	35
Guava	33	3173	44	68	33	145
Guice	34	575	13	22	15	50
Hystrix	92	411	10	15	33	58
j2objc	17	4510	3	10	16	29
Jadx	86	844	7	30	121	158
Java Poet	1	39	1	1	0	2
Jenkins	110	1621	24	102	200	326
JUnit4	64	468	15	10	62	87
Leaky Canary	1	1	0	0	0	0
Logger	4	14	2	0	0	2
Mockito	125	919	6	17	119	142
Moshi	11	86	4	0	2	6
MPAndroidChart	31	223	13	2	22	37
MyBatis-3	231	1235	22	6	112	140
OkHttp	26	152	1	0	11	12
OpenRefine	80	988	1	0	2	3
PhotoView	5	25	1	0	0	1
Picasso	4	33	1	1	0	2
Realm Java	60	629	0	0	0	0
Rebound	10	35	0	0	0	0
Retrofit	28	282	4	5	2	11
Scribe Java	40	285	13	5	19	37
Socket.IO client	8	30	1	0	0	1
Tinker	57	285	13	5	28	46
Zxing	37	499	18	6	22	46
TOTAL	2376	26176	385	381	1223	1989

Table 5.2: BRUNELLESCHI *aggregate output.*

component with a high value of efferent coupling and low value of afferent coupling, *i.e.*, the component depends on a high number of components, and few components are depending on it. This means that the unstable component is likely to change often as a consequence of changes in one or more components it depends on, and so the *instability*. According to Martin, unstable components should address what frequently changes, *i.e.*, GUI, I/O, and many others, while stable components should address high-level concepts and business rules. This is further explained with the *dependency rule*, which prevents more stable components from depending on unstable ones. Unstable Dependency happens when this rule is violated, meaning that no sufficient care was taken when designing the application's structure, with little or no separation between what is *high-level* and should be protected from changes which do not involves requirements or business rules, and what is *detail* and should be easily replaced.

Although being less diffused and not so common as Unstable Dependency, the causes of Cyclic Dependency and Hub-Like Dependency were further investigated as well. Hub-Like Dependency happens where a class has a massive number of both ingoing and outgoing dependencies. Differently from the more infamous code Blob and God Class code smells, which have a significant number of ingoing dependencies with little or no outgoing one, Hub-Like Dependency have both. However, they share a common feature with such smells: they are usually classes

	Cyclic Dependency	Hub-Like Dependency	Unstable Dependency	Total
Min.	0	0	0	0
Max.	63	102	200	326
1st Qu.	1.00	0.0	0.75	2.75
Median	6.50	3.50	13.50	26.00
Mean	10.69	10.58	33.97	55.25
3d Qt.	13.00	10.75	33.00	64.00
Mean by Class	19.14	25.95	-	-
Mean by Package	23.29	-	83.97	-

Table 5.3: BRUNELLESCHI *output summary data*

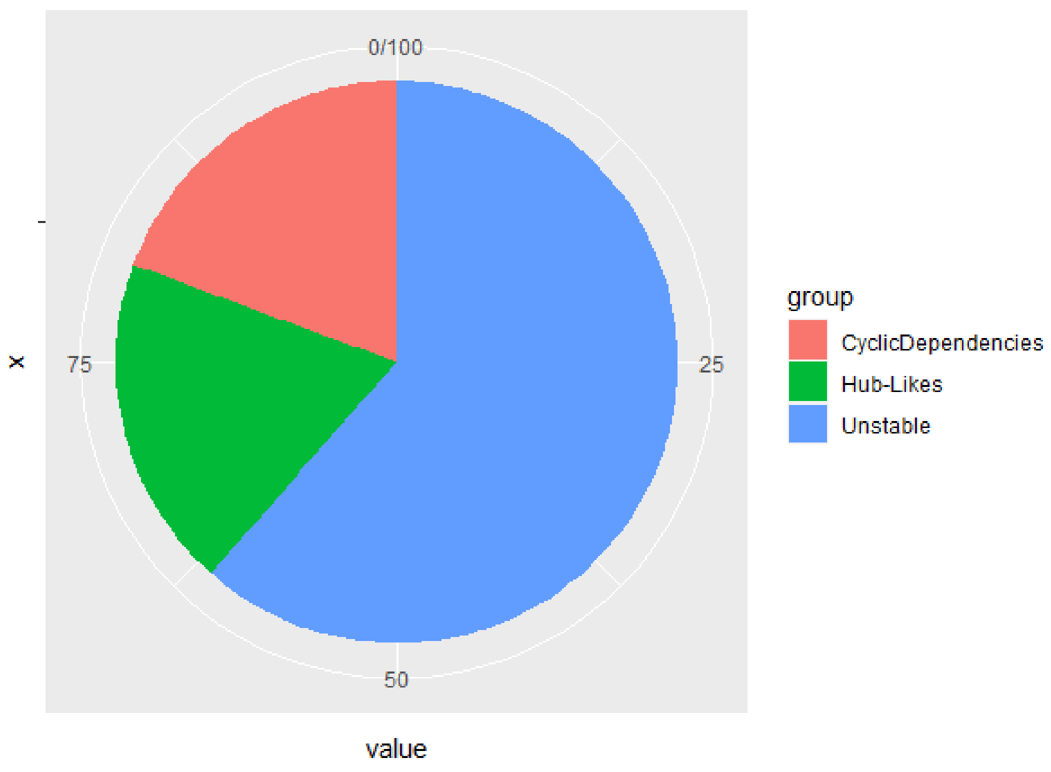


Figure 5.1: Pie chart illustrating the proportions between the found architectural smells

overloaded with responsibilities. It is not already clear why this happens, probably the reasons for their emergence could be similar to their code smell counterparts. Further studies may be necessary to assert this, which may involve historical analysis of the code-bases.

Finally, Cyclic Dependency has also proved to be quite common (they constitute about 20% of the found smell instances). However, their occurrence does not excessively surprise: there are some common practices and patterns that naturally lead to their arising. One of these is the ORM (with or without the usage of libraries). When relationships among entities must be encoded, it is quite reasonable to put in each class a reference to the other, thus creating a -although tiny- cycle. Even some design patters, such as *decorator* or *composite* [20], naturally lead to the occurrence of these kind of cyclic structure. Indeed, this may be another topic of investigation. What keeps to be obscure is what causes the emergence of other topologies, especially larger ones, of Cyclic Dependency.

Cycles at package level can easily be explained considering them as Cyclic Dependency instances that insist on classes that belong to different packages. It is simple to understand that these instances involving more packages are more critical than others since changes in classes residing in a package could trigger an avalanche effect, which eventually involves classes belonging to other packages. What said before only explains part of the problem, since it is still unknown why such classes in different packages are linked together by a cycle. Indeed, also for these cases, further studies will be necessary, although one might suppose that some of them are inducted when careless maintenance is carried out.

Finding 2. *Architectural smells are widely diffused in code-bases: there were found 1989 smells on 36 analyzed projects, with a mean of 55 smells per project. The weighted mean by class and by package was computed as well, resulting of 19.14 Cyclic Dependency per class, 23.29 Cyclic Dependency per package, 25.95 Hub-Like Dependency per class and 83.97 Unstable Dependency per package.*

Tables 5.4 and 5.5 reports the results of the analysis carried out to answer $\mathbf{RQ}_2$. Specifically, they report the co-occurrences of the four analyzed code smell types in the considered projects, both at class and package level. As can be seen, there are co-occurrences among all the smell types occurring at the same level of granularity, and all of them, except for class-level cyclic dependency, are greater than 20% of the instances, meaning that is very likely that one smell instance can imply the presence of another smell.

Concerning the relationship between class level Cyclic Dependency and Hub-Like Dependency, almost every class affected by a Hub-Like Dependency is involved in a Cyclic Dependency. This is not an extraordinary outcome since a class coupled with a significant number of other classes is very likely to be *the crucial point* of a great cycle. For instance, in AndroidUtilCode project, the class UtilsBridge in the package com.blankj.utilcode.util is an instance of Hub-Like De-

	Cyclic Dependency	Hub-Like Dependency
Cyclic Dependency	-	$\frac{351}{3842} \approx 9\%$
Hub-Like Dependency	$\frac{351}{381} \approx 92\%$	-

Table 5.4: Co-occurrences between class smells

	Cyclic Dependency	Unstable Dependency
Cyclic Dependency	-	$\frac{397}{796} \approx 50\%$
Unstable Dependency	$\frac{397}{1223} \approx 32\%$	-

Table 5.5: Co-occurrences between package smells

pendency. Indeed, this class is involved in a great Cyclic Dependency instance, comprising 25 classes, all contained in the same package. Fortunately, only 9% of the classes involved in a Cyclic Dependency are also instances of Hub-Like Dependency. Cyclic Dependency is a smell that affects multiple classes at the same time, and it only requires that each of them had at least one dependency: so it is reasonable to think cycles made up by classes with just one ingoing and one outgoing dependency. Moreover, Cyclic Dependency can also occur among a small number of classes, which is not sufficient to create a Hub-Like Dependency, *e.g.*, cycles between two classes that depending on one another. An example of this can be found in AndroidUtilCode project itself: the classes DialogLayoutCallback and BaseDialogFragment, both contained in the com.blankj.base.dialog form a Cyclic Dependency, since they depend on one another, but indeed none of them is a Hub-Like Dependency.

Concerning the relationship between package level Cyclic Dependency and Unstable Dependency, about half considered packages involved in a Cyclic Dependency are also affected by Unstable Dependency. Although it is not clear yet, there might be several explanations. Since this smell occurs when a stabler component depends on a less stable one, these unstable components have a significant number of outgoing dependency very few ingoing ones. So it is very likely that

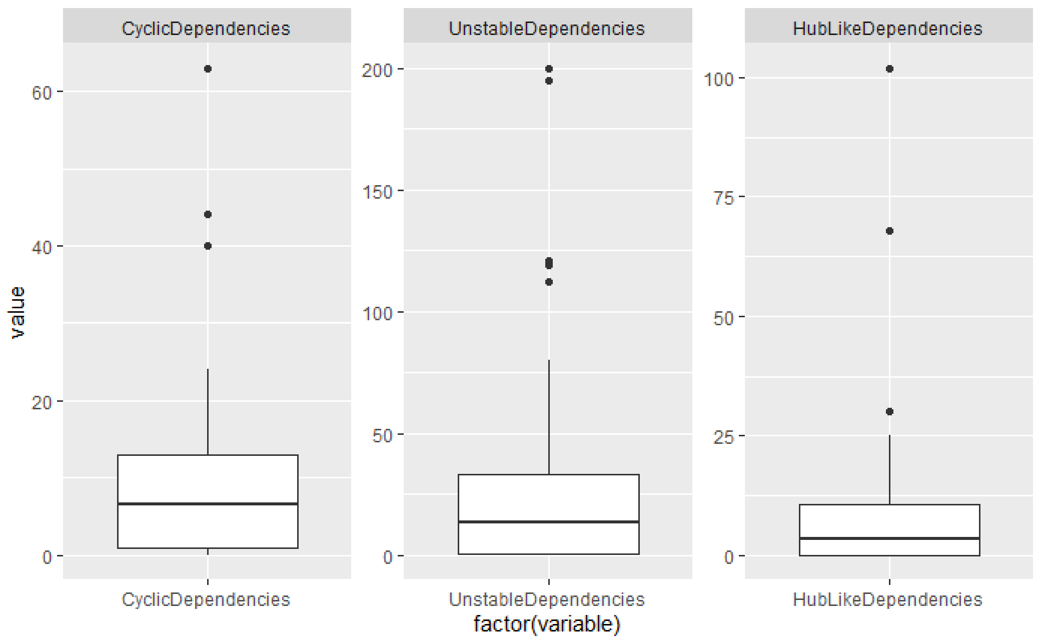


Figure 5.2: Absolute number of code smell instances in the analyzed systems.

these outgoing dependency could create a cycle (both directly and indirectly) with the stabler component. Another explanation might be that both smells are introduced by maintenance activities when a stabler component needs some features implemented in by some classes contained in a less stable one. Carelessly coupling these packages, ignoring both the *Dependency Rule* and the *Acyclic Dependency Rule* defined by Martin [4], not only creates a Cyclic Dependency instance, but also an Unstable Dependency one. However, further studies, such as a historical one, are necessary to assess these hypotheses, since the information gathered so far is not sufficient to determine the reasons for the co-occurrences among these smells.

Finding 3*. Instances of the considered architectural smells often imply the presence of other smells affecting the same component, both at class and package level.*

Threats to validity

In the following section, threats to the study validity are discussed. Threats to *construct validity* are related to the relationship between theory and observation. In the context of this study, they are mainly due to imprecisions or errors in the performed measurements. However, the used tool, BRUNELLESCHI, has proved to be highly reliable: it scored 93% in precision, and it was conjectured that its *recall* is 100%. So, if errors are introduced, they can only be overestimations, *i.e.*, false positive instances, which might be more acceptable than underestimations that might overlook some severe problems.

Threats to *conclusion validity* concern factors that link treatments and outcomes. A threat in this category may be related to the co-occurrences analysis, as frequent co-occurrences between the considered smells might be the results of the high diffuseness of one smell type (*e.g.*, Unstable Dependency) possibly indicating no causation in the observed relationships. However, Figure 5.2 shows box plots depicting the distribution of architectural smell instances for each type present in the dataset. As it is possible to notice, all the smells are almost equally distributed and, therefore, the observed co-occurrences should not be just the result of the high diffuseness of single architectural smell types.

Finally, to what concerns *external validity*, *i.e.*, the generalization of the reported findings, it is known that 36 projects might seem insufficient to be a representative sample. However, on the one hand, a large scale analysis has already been planned, and on the other hand, it is worth saying that the considered projects present a great variety of characteristics, both in domain and size, despite being solely Java systems. Moreover, although the considered smells might not be sufficient in number, it must be said that these are the only ones that currently can be measured by currently available open-source tools, and thus nothing more can be made until the support for other smell detection is provided. For this reason, the generalizability of these results is limited to the four smell types considered.

5.2.2 Qualitative Study Outcomes

In this section, findings for $\mathbf{RQ}_3$ and $\mathbf{RQ}_4$ are presented and discussed, analyzing the responses to the survey. Table 5.6 present the demographic information of the survey participants. Currently, only 11 people have responded. The first thing that can be noticed from Table 5.6 is that most of the respondents are students and have different degrees. Most people also have less than five years of experience. There are two researchers and just one (voluntary) professional developer as well. Figure 5.3 depicts the given answers for the questions about the understandability of the Cyclic Dependency definition. As the left part of the figures shows, most of the participants rated to 5 (highly understandable) the definition of Cyclic Dependency, but only 45.5% of them think that it is a comprehensive description of the problems. This is a first clue of the fact that architectural smells lack of a clear-cut definition. Further analyzing the answers given to the first question ("What is your definition of 'software architectural problem'?"), most of them are quite generic or confused, meaning that the respondents have little awareness about these problems. Furthermore, only two of them answered that the definition *do not* depicts a complete picture of the problem, while the other was not sure.

Education			Experience (years)		
High School or eq.	2	18.2%	1-4	5	45.5%
Bachelor	6	54.5%	5-8	4	36.4%
Master	1	9.1%	9+	2	18.2%
Ph.D.	2	18.2%			
Current Employment			**Team size**		
Student	8	72.7%	1-5	8	72.7%
Researcher	2	18.2%	6-10	3	27.7%
Vol. Dev.	1	9.1%			

Table 5.6: Demographic information of survey participants

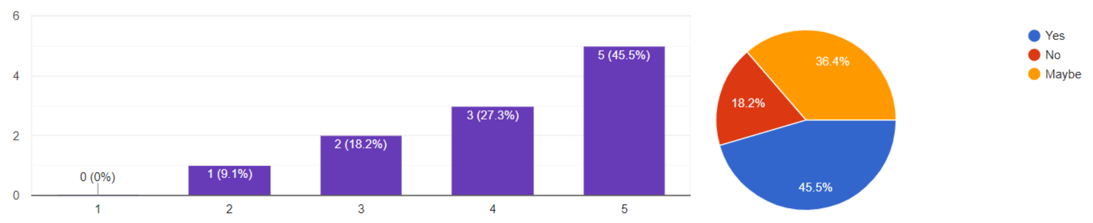


Figure 5.3: Responses to the Cyclic Dependency understandability question

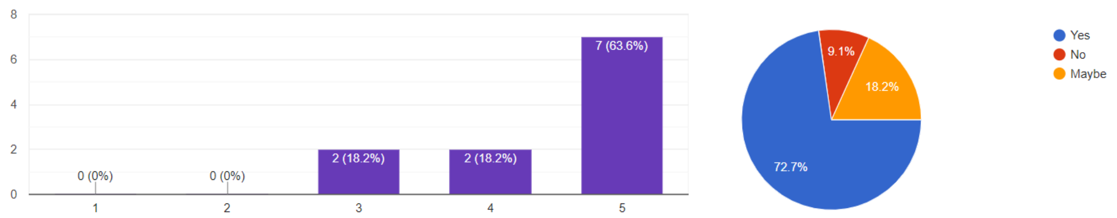


Figure 5.4: Responses to the Unstable Dependency understandability question

These two also responded the optional question about the reasons they do not believe that the definition was not complete: one of them answered that "Should be specified the degree of the dependencies", and the other answered that it is not enough to know that components in a Cyclic Dependency depend on each other to identify an issue.

However, the other similar questions have received entirely different answers, as most of the participants recognized the given definitions as clear and complete, as shown in Figure 5.4. Results become more curious than before when analyzing the answers about the impact of the smells on understandability and maintainability. On the one hand, most of the answers indicate that these smells might have a significant impact on both of the considered properties, however, on the other hand, little or no answers were given when asking to deepen the reasons of the given score. The given answers show that the real reasons for the impact of these smells on the considered properties were not completely caught: this is particularly evident for Unstable Dependency, while for Cyclic Dependency, there is a reasonable degree of awareness. Although, as shown in Figure 5.4 most of the participant reckoned that the definition was clear and complete, when depicting the reasons of the impact of Unstable Dependency on understandability and maintainability, it is evident that they do not understand what *unstable* actually means. Finally, despite the 66.7% of the participants reckon that an automated architectural smells detection tool could be useful, just two of them (the researchers) said in which phase of software development they would use it: one in a Dev-Ops pipeline, after the testing phase, and one in development and maintenance phases.

Finding 4*. Developers are not fully aware of what architectural problems are and to what extent they impact code-bases quality. Although they perceive the potential usefulness of an automatic detection tool, they do not know in which development phase it might be useful.*

Threats to validity

It is known that the currently received answers lack *external validity*, as they are just 11. Another threat of this kind is that the conducted sampling, which is a convenient one, on the one hand, was chosen to widen as much as possible the answers received, but on the other hand, it could be not representative of the entire population, since the majority of the respondent were students. However, a more substantial, systematic study has already been planned, aimed to address the same RQs faced by this one. Moreover, it has been planned to survey the communities that have developed the analyzed projects to assess whether they are aware of the architectural problems living in their code-bases, and how grave they perceive these issues. Also, it is worth to say that since most of the received answers come from students, there might be a threat to *internal validity*, as the participants might be afraid of being evaluated, and may not have reported their true deviations between estimate and outcome, but some false but ”better” values.

Chapter 6

Conclusions and Future Works

In this chapter, a summary fo the work conducted so far is carried out, and a proposal for future fork is stated.

6.1 Summary

In this thesis, the problem of architectural smells has been faced in many facets. First of all, it has been shown that architectural smells are problems that have received less attention both in academia and industry than their more infamous *cousins*, *i.e.*, the code smell. Despite this, efforts have been spent in both defining and detecting them. Several tool, presented in Chapter 2 have been developed to allow developers to detect architectural smells in their code-bases, such as ARCAN [6] or DESIGNITE [7]. However, most of them are not currently available or released under a commercial license, limiting both researchers and practitioners in finding and resolving these flaws. On the other hand, efforts were also spent in theoretical issues, since most of the smells lack a clear-cut definition, which limits the practitioners' awareness and perception of these.

To overcome all of this, in this thesis, a novel open-source automatic architectural smells detection tool was presented, able to spot three kinds of architectural smells (Cyclic Dependency, Hub-Like Dependency, Unstable Dependency) both at class and package level. These three smells were chosen since they are the ones with the most explicit definition, so it was possible to formulate detection rules directly from it, with no need to invent heuristics. Besides, a validation experiment was conducted, aimed to assess the tool's precision, which is 93%, while its recall was temporary left behind since it was conjectured that it is 100% just because no heuristic was used, and so there is no chance of false-negative instances.

Then, once assured that the tool was reliable and well-performer, it was used to conduct a quantitative study aimed to assess the diffusion of architectural smells in code-bases and their possible co-occurrence. The study was conducted on a set of 36 Java open-source projects coming from GitHub. The results showed that not only these architectural flaws are quite diffused problems, but also that often the presence of one implies the occurrence of another.

Finally, a qualitative study was also conducted, aimed to assess developers' awareness of architectural issues and whether they feel the need of an automatic architectural smells detection tool, and in which development stage they would use it. A survey was conducted worldwide, but at the time of writing, only 11 responses were received, mainly coming from students. Although these findings lack a robust external validity, some initial considerations can be asserted: as quite expected, despite they claim to understand the proposed problems, it is evident that they are not sure of what actually an architectural flaw is. Moreover, although they reckon that an automatic architectural smells detection tool might be useful, they are not sure in which development phase they might use it.

6.2 Lesson Learned and Future Works

The scenario described in the previous section leads to a wide range of options for future work. Many paths are left open, either to what concerns the tools and to what concerns empirical studies.

6.2.1 Brunelleschi's Future Enhancements

To what concern Brunelleschi's future, the most straightforward, and already mentioned, is the construction of a better GUI and a better output format. As said, only a CSV-based output is currently provided, since the tool is in the so-called experimental version. So, a GUI needs to be built before releasing it on a large scale. In the provided GUI, different types of analysis will be allowed, *e.g.*, selecting the desired smell to be analyzed, as well as the choice of output, *i.e.*, visual or CSV-based, will be allowed.

Another needed enhancement, as already said, is in the graph building algorithm, as it proved itself to be extremely inefficient both in terms of time and space. To achieve this, however, a more in-depth study of IntelliJ IDEA API documentation is needed. However, as this documentation is scarce and ill-maintained, it might be needed a significant amount of effort to achieve this, and so it cannot be an immediate enhancement.

Concerning quality and maintainability, so far, it was made the most flexible, change-prone choice, and so the level of indirection among class dependencies is very high. A trade-off could be evaluated in order to simplify the code-base and to reduce the complexity. For instance, at the moment, to obtain a smell detector, an instance of the builder must be called, and then a command is constructed and passed to the interactor, which will execute it. A possible simplification might be to make the command directly assemble the chosen detector and then run it. This would make the introduction of a new smell easier since it would require a smaller number of modifications.

Indeed, there are various smells whose detection could be supported pretty straightforwardly. Directly relying on the dependency graph, two other architectural smells can be supported, *i.e.*, Unutilized Abstraction and Abstraction without Decoupling, since their detection rules easily can be verified using the graph representation. However, other kinds of smells, need historical analysis on multiple versions of the code-base, so graph-based rules would not be suitable, and the implementation of several new modules would be necessary. Finally, some other smells might not be easily implemented because the ambiguity of their definition would not make it easy to define detection rules.

6.2.2 Further Empirical Studies

As already said, there are a wide plethora of further studies that can be conducted. It was learned that architectural smells are widely diffused in code-bases, and often the presence of one smelly instance implies the presence of others, however, despite some reasonable deductions that have been presented during the results' discussion, there is no sufficient data that allows understanding the root causes of the smells' arising and their co-occurrences. To this extent, the first study to be conducted should be a historical one that might make explicit the causes of the relationships among different architectural smells and when they are introduced. Also, insisting on the causes that lead to architectural smells instances proliferation, another empirical study might be conducted, aimed to assess the what extent design patterns and common practices are related to the presence of architectural smells instances in code-bases.

To what concerns qualitative studies, although the answers to the presented survey were very scarce, they pointed out that very few developers (both researchers and practitioners) are aware of what architectural smells are and to what extent they could affect the overall quality of software projects, and indirectly affect the cost and the efforts needed to maintain them. To the aim of deepening and assessing the developers' perception of this kind of problem, a more extensive study aimed may be conducted. Another survey may also be conducted to assess if the communities that have developed the analyzed projects are aware of these kinds of problems, how they perceive them, and whether they would solve them. Furthermore, as it has been done for code smells, a study aimed to assess the relationship among architectural smells and community types and smells can be set up to understand whether social debt may cause technical debt at this level of abstraction as well.

Bibliography

[1] M. Fowler, K. Beck, J. Brant, and W. Opdyke, "Refactoring: improving the design of existing code. 1999," *Google Scholar Google Scholar Digital Library Digital Library*.

[2] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 403–414, IEEE, 2015.

[3] N. Medvidovic and R. N. Taylor, "Software architecture: foundations, theory, and practice," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, pp. 471–472, IEEE, 2010.

[4] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* USA: Prentice Hall Press, 1st ed., 2017.

[5] U. Azadi, F. A. Fontana, and D. Taibi, "Architectural smells detected by tools: a catalogue proposal," in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pp. 88–97, IEEE, 2019.

[6] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan: A tool for architectural smells detection," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 282–285, IEEE, 2017.

[7] T. Sharma, “Designite: a customizable tool for smell mining in c# repositories,” in *10th Seminar on Advanced Techniques and Tools for Software Evolution, Madrid, Spain*, 2017.

[8] M. Lippert and S. Roock, *Refactoring in large software projects: performing complex restructurings successfully.* John Wiley & Sons, 2006.

[9] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, “Identifying architectural bad smells,” in *2009 13th European Conference on Software Maintenance and Reengineering*, pp. 255–258, IEEE, 2009.

[10] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, “Toward a catalogue of architectural bad smells,” in *International conference on the quality of software architectures*, pp. 146–162, Springer, 2009.

[11] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic, “Relating architectural decay and sustainability of software systems,” in *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 178–181, IEEE, 2016.

[12] S. Ganesh, T. Sharma, and G. Suryanarayana, “Towards a principle-based classification of structural design smells.,” *J. Object Technol.*, vol. 12, no. 2, pp. 1–1, 2013.

[13] R. Mo, Y. Cai, R. Kazman, and L. Xiao, “Hotspot patterns: The formal definition and automatic detection of architecture smells,” in *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pp. 51–60, IEEE, 2015.

[14] Logarix, “Ai reviewer.” `www.aireviewer.com`. Accessed: 2020-03-05.

[15] R. C. Martin, *Agile software development: principles, patterns, and practices.* Prentice Hall, 2002.

[16] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, “Are automatically-detected code anomalies relevant to architectural modularity? an exploratory analysis of evolving systems,” in *Proceedings of the*

11th annual international conference on Aspect-oriented Software Development, pp. 167–178, 2012.

[17] A. Martini, F. A. Fontana, A. Biaggi, and R. Roveda, "Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company," in *European Conference on Software Architecture*, pp. 320–335, Springer, 2018.

[18] H. N. Gabow, "Path-based depth-first search for strongand biconnected components; cu-cs-890-99," 1999.

[19] I. Sommerville, "Software engineering 9th edition," *ISBN-10*, vol. 137035152, p. 18, 2011.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of reusable object-oriented sofware.* Addison Wesley, 2009.

[21] "Dsm analysis." `https://www.jetbrains.com/help/idea/dsm-analysis.html`. Accessed: 2020-03-05.

[22] "50 top java projects on github." `https://medium.com/issuehunt/50-top-java-projects-on-github-adbfe9f67dbc`. Accessed: 2020-03-05.