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 work a novel tool is presented, Brunelleschi, which can detect 3 types of smells at different level of granularity, which are Cyclic Dependency, detected on both class and level package, Hub-Like Dependency, detected on class level, and Unstable Dependency, detected on package level. The tool is then validated and used in the context of a family of empirical studies, aimed to better understand architectural smells diffusion and characteristics.

Contents

Chapter 1

Introduction

In software engineering world it is known that the longest part of a software life is in the maintenance phase. Once delivered to the customer, in fact, it will be necessary, over time, to proceed with continuous and periodic changes in order to preserve its usefulness. To this extent, Lehman's first law clearly explain 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 an system evolves, its complexity increases unless work is done to maintain or reduce it."

While on one hand, results of a careless or not well conducted maintenance activity have been objects of a great number of studies, since Martin Fowler published his book on *refactoring* [5], on the other much more dreadful and underestimated problems, happening at architectural level, have been overlooked. 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 erode the code maintainability and understandability [6]. 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.

Unfortunately, this is not the only kind of technical debt. Sub-optimal decision, and bad choices might occur also in designing the architecture of a software, leading to the introduction of another type of smells, *i.e.*, architetcural 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 architetcural smells, is called "architectural drift" (Medvidovic *et al.* [3]). When the deviation starts to worsen, and indeed introducing a lot of sub-optimal solutions, is called "architectural erosion" [3]. Erosion must be dealt in early stages, otherwise it will lead eventually to the aforementioned problems. To face this, a work of "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 an higher level of abstraction also means that the impact of architetcural 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 architetcural smell is way harder that a code smell. Although their *dreadfulness*, architetcural 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 [7], but also, automatic tools able to detect them have been developed, both in academia and industry [8, 9].

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 work. 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 on both class and level package, Hub-Like Dependency, detected on class level, and Unstable Dependency, detected on 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 works about architectural smell are explored, in particular, a definition and an overview of the most known architetcural smells is presented. Chapter 3 focuses on Brunelleschi, its features, its architecture, a usage scenarios and some future enhancements. In Chapter 4 the empirical studies will defined, as well as their design and in Chapter 5 their results and validity will be discussed. Finally, Chapter 6 wraps everything up and a presents some possible future works.

Chapter 2

Related Works

Chapter 3

Brunelleschi: A Novel Tool

In the previous chapter, there were presented all the efforts done so far to tackle architetcural smells. However, there are some limitations, both theoretical and practical. The former is that, despite what has been proposed, some smells still lack a clear-cut definition, and thus, some boundary cases are not properly managed. Furthermore, smells instances could not easily be classified in one definition or another. This is still an open challenge, that requires a better understanding of architetcural smells. About the latter, 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 architetcural 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 architetcural smells in Java projects, *i.e.*, Cyclic Dependency (both on class and package), Hub-Like Dependency (on class level), and Unstable Dependency (on package level). In following sections a comprehensive description of Brunelleschi is carried out: Section 3.1 proposes a general overview of the tool; Section 3.2 documents the architectural choices made and implemented; In Section 3.3 the tool validation procedure is documented and finally, in Section 3.4 a proposal of future enhancements is discussed.

3.1 Overview

Brunelleschi, as said several times before, is an automatic architetcural smells identification tool for Java projects, integrated and deployed directly in IntelliJ IDEA, able to detect 3 architetcural smells in source code, namely Cyclic Dependency, Hub-Like Dependency, Unstable Dependency (made explicit in Chapter 2). For each of these architetcural smells a detection rule was constructed, deriving it directly from the smell definition [7], 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 dependencies 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 architetcural smells candidates. The mentioned rules are here carried out.

Cyclic Dependency Cyclic Dependency is detected applying Gabow's algorithm presented in Path-based depth-first search for strong and biconnected components by Gabow [10], and it is run on both class and package dependency graph. Since in a directed graph a strongly connected component, to be so, requires that the nodes within it are linked thorough at least one cycle, finding them allows Brunelleschi to spot all the software components involved in any of them.

Hub-Like Dependency Hub-Like Dependency smells 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 [9], and the one that is also reported in the catalog [7], which has proved itself to be valid. However, it worths saying that when building the graph, the tool does not take into account dependencies from external libraries, *e.g.*, Java standard

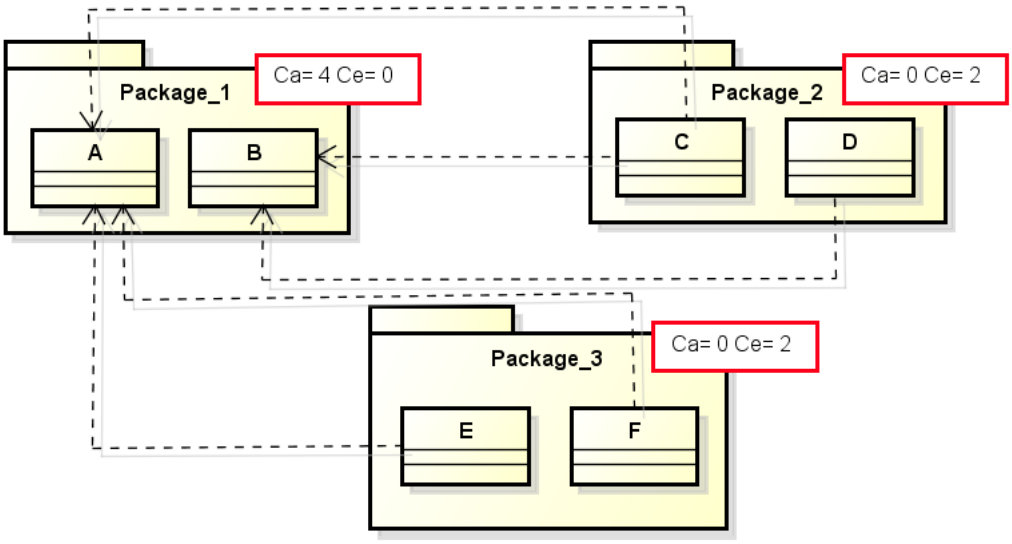

Figure 3.1: Example of Instability Metric calculation

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 [8] faced, does not subsists.

Unstable Dependency Unstable Dependency is a smell that directly come 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 3.1. Then, the instability of each successor node (namely a node reached by an outgoing edge, *i.e.*, a package which 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 having performed these analyses, Brunelleschi gives in 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 smell, 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 3.4, 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 worths spending some words about the used notation: following to Ian Sommerville [11], 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.

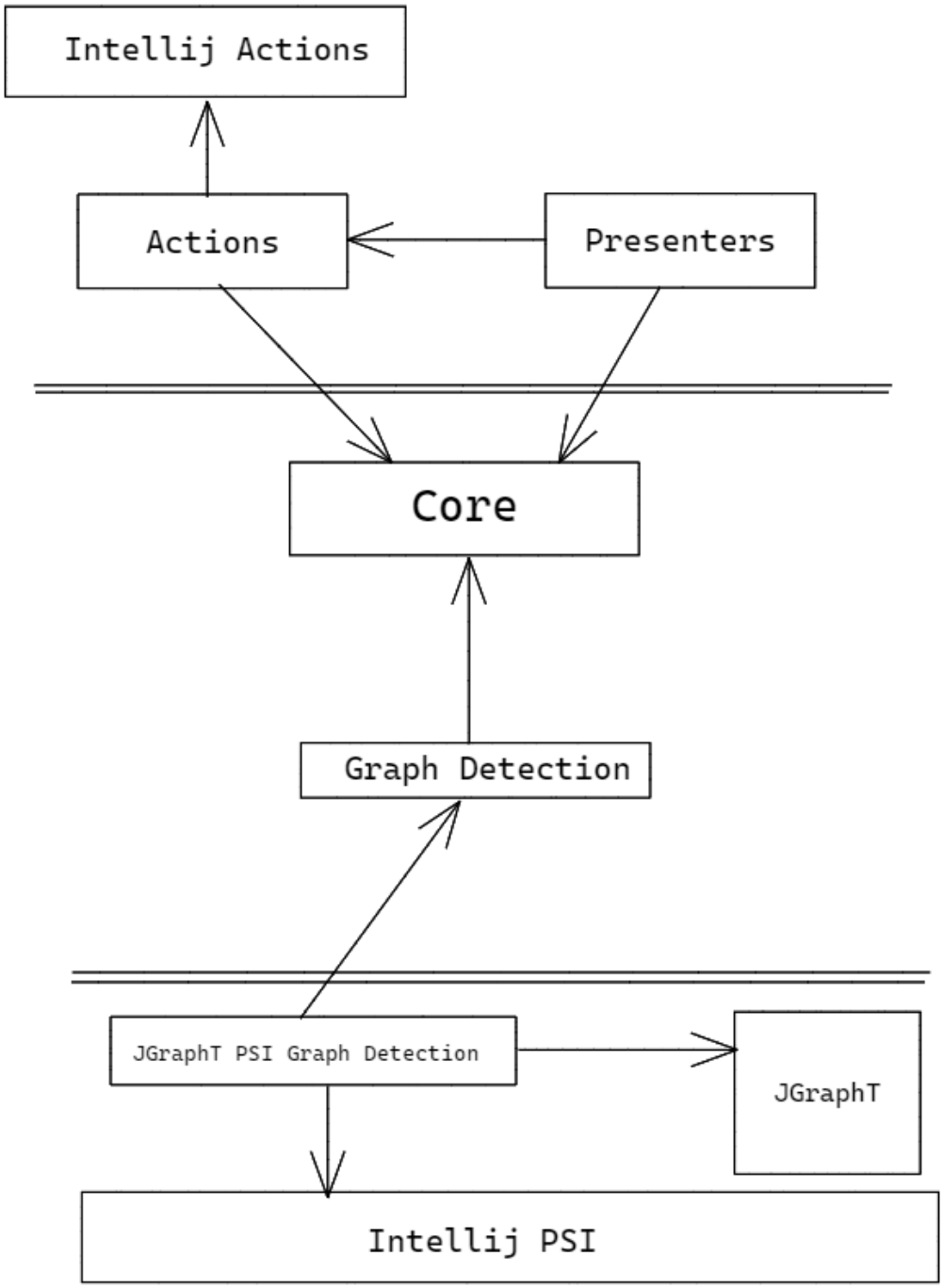

Figure 3.2: Representation of Brunelleschi's High-Level Components

If this might seem a bit confusing, Figure 3.2 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 Martin intended it: 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 low-level details, such as GUI or Databases or I/O devices. To make this happen, the *Dependency Inversion Principle* and *Open/Closed Principle* [12], 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, following in this section, a description of each showed component is carried out, specifying the design decision made, and with this, all the *plug-in matter* will become clear.

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 [13], in order to combine at runtime different kinds of analyses. To construct it, the builder pattern [13] 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 computation 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. Here resides a dependency graph concrete implementation, which relies on a graph library, namely JGraphT (as the figure shows). Here are also located 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 needed to be used, or a different algorithm for dependency class construction, either using Psi or not, 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, in what traditionally is called *presentation layer*, resides the Presenter component. As the name suggests, here live 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 takes only 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.

So, in light of this, 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 does not depend from anything, one could easily extract them 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 library are easily reusable. Keeping available any kind of future changes, even a *divorce* from IntelliJ IDEA, is clearly the most important design decision made.

3.3 Validation

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

RQ$_0$. *What is* Brunelleschi*'s precision in detecting architetcural smells?*

The context of the study is made up by 37 Java open source projects taken from GitHub, whose characteristic are resumed in table 3.1.

#	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 3.1: Basic characteristics of the software project considered

3.3.1 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 architetcural smells (385 Cyclic Dependency, 381 Hub-Like Dependency, 1223 Unstable Dependency). These results 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 architetcural smells.

The Cyclic Dependency instances validation relied on the Dependency Structure Matrix (DSM) provided by IntelliJ IDEA itself [14]. DSM a method that helps developers 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 3.3, 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.

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, DMS also fitted in the Unstable Dependency instances validation. 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

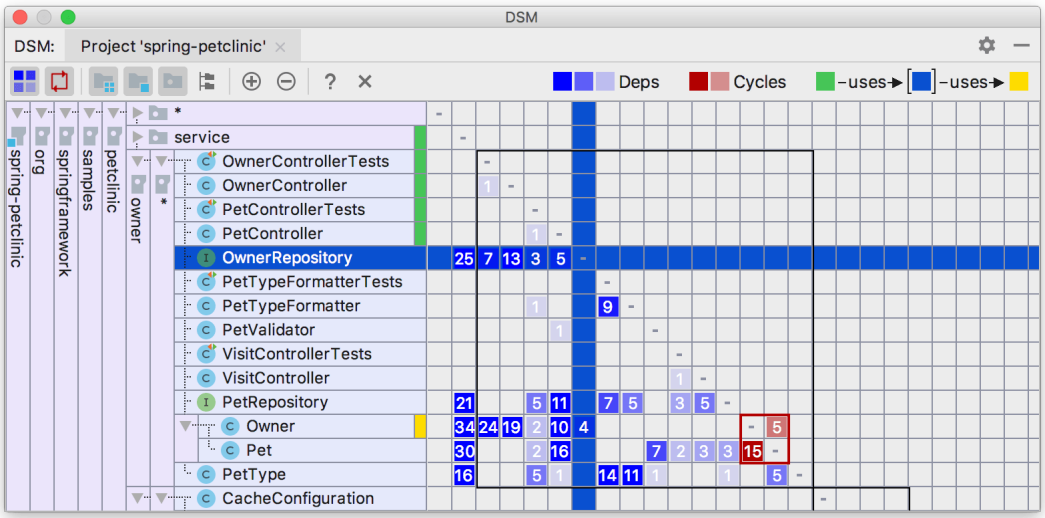

Figure 3.3: An example of DSM tool window

easy to evaluate if a candidate actually depended from a more unstable component.

3.3.2 Results

MANUEL ▶*La tabella con i valori finali di TP e FP deve essere ancora realizzata, così come il calcolo della precsion finale. Tuttavia, dalle analisi risulta che solo per uno smell si ha un valore inferore a 100%*◀ Table Y shows the obtained results, both global and for each specific smell. It worth noting that only for Cyclic Dependency a precision score lower than 100 was calculated. So, a further investigation was conducted and it showed that only class-level instances showed the presence of false positive, while package-level instances none. 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 this is still not clear, why some *false* edges may appear, this is may caused by the fact that annotations and interfaces are considered as normal classes, and this

might lead to the definition of false edges. However, since the error is limited, and there is only a lack of precision, it could be not 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 ulterior reason to either fix or replacing it.

Finding 1. BRUNELLESCHI *scored a X precision.*

3.3.3 Discussion

BRUNELLESCHI has proved itself to have a good precision score. However, recall was neither considered nor evaluated. This was not left to choice, on the contrary, it 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 [7]. So, one could consider them as methods' specification, and thus, what they actually needed was a testing and not a validation session, which testing was actually performed. Considering the case of an actual smell not detected by BRUNELLESCHI, there must be only one possible reason to explain it, and is 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 a unspotted bug, or that the recall is maximum. However, this is the classical testing problem, and nobody could be 100% sure that its software is bug-free. So, as far as it is known, the only possible scenario is the one with maximum precision.

Finding 2. *Since* BRUNELLESCHI *is a rule-based and not a heuristic-based tool, its recall score is 100%.*

3.4 Future Enhancements

In this section some immediate future enhancement are discussed. 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 large scale. In the provided GUI, not only different type of analysis will be allowed, *e.g.*, selecting the desired smell to be analyzed, but also 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 deeper study of IntelliJ IDEA API documentation is needed. However, as these documentation are really scarce and ill-maintained, it might be needed a great amount of effort to achieve this, and so it cannot be an immediate enhancement.

Speaking about 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 it is constructed a command and passed to the interactor, which will execute it. A possible simplification might be to make the command directly assemble and then run the detectors. This would also make the introduction of a new smell easier, since it would require a smaller number of modifications.

Speaking about that, there are various smells whose detection could be supported pretty straightforwardly. Directly relying on dependency graph, two other architetcural smells can be supported, *i.e.*, Unutilized Abstraction and Abstraction without Decoupling, since their detection rules can easily be verified using the graph representation. Other kind of smells, however, need historical analysis on multiple versions of the code-base, and so, to provide it, graph-based rules would not fit, and the implementation of several new modules would be necessary. Finally, some other smells might not be easily implemented, because their definition itself leaves for ambiguity, and it could not be easy to encode them in a detection rule.

Chapter 4

Design of the Empirical Studies

This chapter focuses on the design of the three empirical studies that were carried out to widen the knowledge about architetcural 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 architetcural smells with the *purpose* of better understanding their diffusion, their relationships and how the developers are aware of their existence in code-bases, and if they need an automatic tool. The perspective is of both researchers and practitioners, who are interested in discovering the potential issues caused by architetcural smells. impact To achieve this goal, three research questions are explored:

RQ$_1$. *How much architetcural smells are diffused in software projects?*
RQ$_2$. *Does an architetcural smell occurrence imply the presence of another?*
RQ$_3$. *As far as they know about architetcural smells, do developers need an automatic tool for architetcural smells detection?*

In section 4.1 the description of how the first two RQs were addressed, while Section 4.2 carries out the design of the qualitative study.

4.1 The quantitative studies

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

4.1.1 Projects

For these experiments was employed a dataset of 37 Java Open-Source projects present on GitHub, 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 this article [15]. The number of analyzed projects was then reduced to 37 as at the time of the analyses, their builds were not passing and so it was not possible to compile them, and so not allowing Brunelleschi to run. 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 (*i.e.*, the number of lines of codes divided by one thousand). While the latter is quite understandable, as this conveys a greater generalization of the experiments' outcomes, the former needs some explanations: it is quite expected to find that the majority of the considered projects were libraries, as the choice of them comes form popularity (stars on GitHub). However, even complete softwares were found, although being tools, in most cases. For the same reason, also some frameworks were taken into account.

4.1.2 Architectural Smells

The choice of the architetcural 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

#	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

validation study, discussed in Chapter 3, showing a precision score of X and so it can be considered fairly reliable. In addition, these smells are the one with a clear-cut definition [7], and thus the one that could more easily investigated in code-bases. Moreover, since their simplicity, they are a good candidate for a qualitative analysis among developers.

4.1.3 Data Analysis and Collection

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

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

The first step is pretty straightforward: Brunelleschi was run on the projects, and the results where gathered. Then, the summary files for each projects were aggregated in a single file, containing the detections for all the projects. After that, the so gathered data was analyzed using basic statistic instruments (mean, median, box-plots, *etc.*) exploiting the tool R. It worths to say that not only it was analyzed the absolute number of smells in a software project, but it has also been weighted with the project's total number of classes and packages, to have a relative count. Finally, the third step is aimed to asses how often the presence of a architetcural smell 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 architetcural smell type as_i it was computed the percentage of times its presence in a class/package co-occurs with the presence of another architetcural smell type as_j. Formally, for each pair of architetcural smells as_i, as_j, the percentage of co-occurrences were computed with the following formula:

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

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 $co\text{-}occurrences_{as_{i,j}}$ differs from $co\text{-}occurrences_{as_{j,i}}$, as the formula's denominator changes from $|as_i|$to $|as_j|$.

4.2 The qualitative study

The third research question was addressed through a qualitative study, conducted as a large scale survey. The survey is made up by three sections, and the complete set of questions can be found in table Z. The goal was to asses the potential need and utility of an automatic architetcural smells detection tool, in an indirect way: asking to developers about how aware they are of architectural problems, and to what extent they could affect their code-bases. Then, the most important question is asked in a direct way, of course, and that is 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 from a score of 1 (poorly understandable) to 5 (high understandable) the understandability of those definition, and possibly to provide some further detail about the reason they rated as poorly understandable the definition.

The second section is about the impact and the emergence of architectural problems. For each of the considered smells, the participant 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). Then, they were 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* informations are asked, such as the participant's gender, the title of study, their current working position and the size of their current working team.

The survey was constructed using Google Form service and then spread to de-

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 4.2: Questions presented on survey

velopers' community through social networks and personal acquaintances. There were considered both students and practitioners, in order to have a wider spectrum of knowledge. To this aim, the personal programming experience was considered, and, if necessary, considered as a co-factor in results' analysis.

Chapter 5

Results

In this chapter it is carried out a discussion about the outcome, and their validity, of the experiments described in the previous chapter. In Section 5.1 a discussion on the quantitative studies outcomes is carried out, while in section 5.2 are presented and discusses the qualitative study outcomes.

5.1 Quantitative Studies Outcomes

In this section the findings for $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ are presented and discussed. Table 5.1 shows the row data coming from Brunelleschi analysis on the projects dataset. The table shows a total amount of 1989 architetcural smells found. What already emerges from this table is that no project is completely smell-free, with the exception of few *noble* ones: only 3 out of 37 projects show no instances of any of the considered smell. This means that architetcural smells are a fairly diffused problem.

Looking at Table 5.2 it is possible to note de distribution of the architetcural smells in the code-bases. As noticeable in the first table, the minimum number of found smell is 0, and the maximum is 326 (in a single project). What is also possible to see in this table is the mean number of smell per project, that is 55. This is a fairly large number, that indeed confirms the thesis that architetcural smells are a common and unnoticed problem. However, this mean value is an

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 5.1: Brunelleschi *aggregate output.*

	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 5.2: Brunelleschi *output summary data*

absolute value, not considering the differences between large and small projects. This is why also the weighted mean by package and classes was calculated, considering the different levels of granularity of the smells instances. What comes out is that the means are slightly increased, meaning that the found problem is even more diffused than expected. Looking at both the table, it is possible to deduct that the most diffuse smell is Unstable Dependency, with a 61% of occurrences on the total number of smells. 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 of that could be addressed by recalling both Martin's definition of Instability and Dependency Rule [4]. An *unstable* component is a component with a great value of efferent coupling and low value of afferent coupling, *i.e.*, the component depends on a great number of components and there are few components depending on it. This means that the unstable component is likely to change often in consequence of changes in one or more components which it depends on, and so the *instability*. According to Martin, unstable components should address what it changes frequently, *i.e.*, GUI, I/O and so on, while stable components should address high level concepts and business rules. This is further explained with the *dependency rule*, which prevents more stable components to depend on unstable ones. Unstable Dependency happens

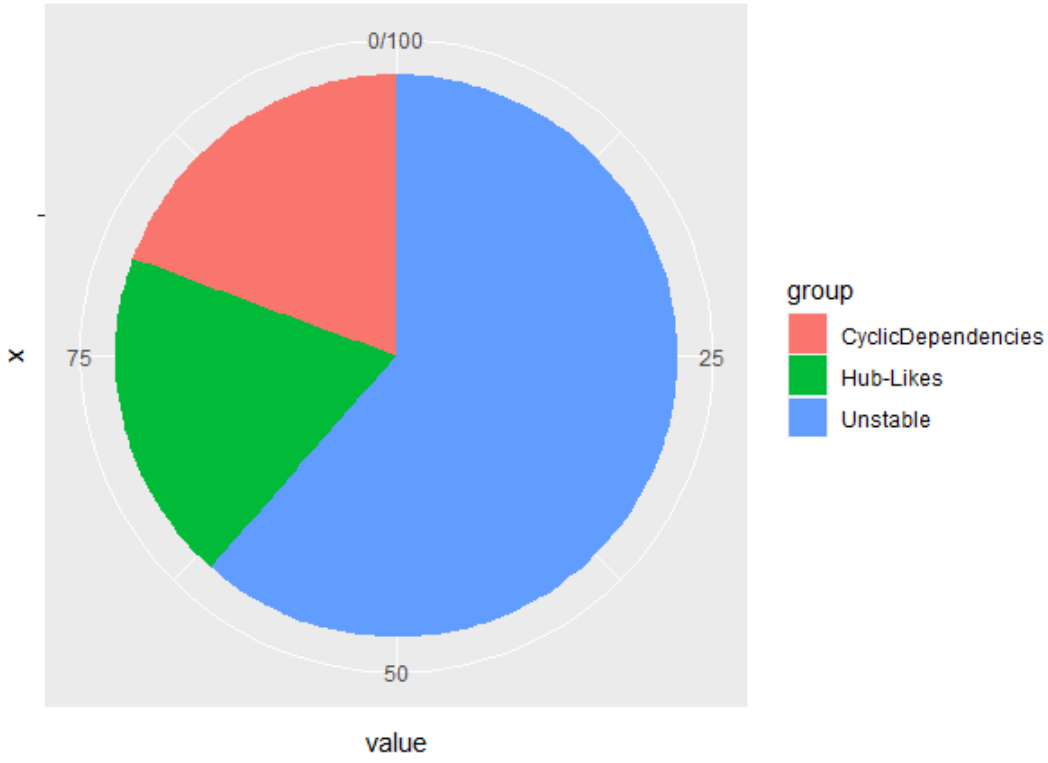

Figure 5.1: Pie chart illustrating the proportions between the found architetcural smells

when this rule is violated, meaning that no sufficient care was addressed when designing the application's structure, with little or no separation of what is *high-level* and should be protected from change which does not involves new or changed requirements or business rules, and what is *detail* and should be easily replaced.

Although being less diffuse and not so common as Unstable Dependency, also Cyclic Dependency and Hub-Like Dependency causes where further investigated. Hub-Like Dependency happens where a class has a huge number of both ingoing and outgoing dependencies. Differently from the more infamous code Blob and God Class code smells, which have a great 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 overloaded with responsibilities. It is not already clear why this happens, probably the reasons of they emergence could be similar to their code smell counterparts. Further studies may be necessary to assert that, which may involve historical analysis of the

code-bases. Finally, Cyclic Dependency have also proved themselves to be quite common (they constitute about 20% of the found smell instances). Their occurrence, however, does not excessively surprise: there are some common practice 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 normal 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* [13], 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 viewing them as Cyclic Dependency instances that insist on classes which belong to different packages. It is simple to understand that these instances, just because involve more packages, are more critical than other, since changes in classes residing in a packages could trigger an avalanche effect which eventually involve classes belonging to other packages. This only clarify part of the problem, since it is still unknown why such classes in different packages are linked together by a cycle. Indeed, also fo these cases, further studies will be necessary, although one might suppose that some of them are inducted when a careless maintenance intervention is carried out.

Finding 3. *Architectural smells are widely diffused in code-bases*

MANUEL ►*Per RQ2 ancora non sono pronti i risultati*◄

5.2 Qualitative Study Outcomes

MANUEL ►*Per RQ3 ancora non sono pronti i risultati, poichè il survey è aperto fino al 30 giugno*◄

Chapter 6

Conclusions and Future Works

Bibliography

[1] P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford, "Documenting software architectures: views and beyond," in *25th International Conference on Software Engineering, 2003. Proceedings.*, pp. 740–741, 2003.

[2] "Software architecture guide." `https://martinfowler.com/architecture/`. Accessed: 2020-06-21.

[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] 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*.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

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

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

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

[15] "50 top java projects on github." `https://medium.com/issuehunt/50-top-java-projects-on-github-adbfe9f67dbc`. Accessed: 2020-03-05.

[16] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pp. 207–216, 1993.