Automatic Detection of Instability Architectural Smells

Francesca Arcelli Fontana*, Ilaria Pigazzini†, Riccardo Roveda* and Marco Zanoni*
Università degli Studi di Milano - Bicocca, Milan, Italy
Email: {arcelli,riccardo.roveda,marco.zanoni}@disco.unimib.it*
i.pigazzini@campus.unimib.it†

***Abstract*—Code smells represent well known symptoms of problems at code level, and architectural smells can be seen as their counterpart at architecture level. If identified in a system, they are usually considered more critical than code smells, for their effect on maintainability issues. In this paper, we introduce a tool for the detection of architectural smells that could have an impact on the stability of a system. The detection techniques are based on the analysis of dependency graphs extracted from compiled Java projects and stored in a graph database. The results combine the information gathered from dependency and instability metrics to identify flaws hidden in the software architecture. We also propose some filters trying to avoid possible false positives.**

***Index Terms*—Architectural Smells, Design Metrics, Software Architecture Evaluation**

I. Introduction

When we must evaluate software quality, we can consider different features, for example code smells, architectural smells, and we can compute a large number of different kinds of metrics. Code smells are symptoms of possible problems at code level that can be removed through different refactoring techniques [1] and can be used as surface indicators of design problems [2], while architectural smells can derive from commonly used architectural decisions, intentional or not, that negatively impact internal software quality [3] with large effects on software maintainability [4].

For what concerns code smells and metrics, many tools have been developed, both open source and commercial ones. Architectural smells (AS) received less attention, even if some tools have been developed, as we outline in Section II.

In this paper, we describe an architectural smells detector tool that we are developing with a focus on the detection of three AS, that we call *instability architectural smells*.

An Instability metric was proposed by Martin [5] and measures the instability of packages, where stability is measured by calculating the effort to change a package without impacting other packages within the application. For instability architectural smells we mean architectural smells that have an impact on the Instability metric. Hence, if we detect this kind of smells we could assist to an increment on the value of this metric with an effect on software evolution and maintainability. In this category of smells, we have identified three smells on which we focus our attention: Unstable Dependency, Hub-Like Dependency and Cyclic Dependency. The first one is an AS detected at package level, the second at class level, and the last one is detected both at class and package level. All these smells are detected by the tool we have developed, called Arcan, through an evaluation of different dependency issues as described in Section III. We show the results we obtained on seven systems. Among the three AS, two AS have been detected by other tools using different techniques. Here, we adopt a technique that, according to our knowledge, has not been applied for AS detection before. The technique exploits graph databases to perform graph queries. This allows higher scalability in the detection, allowing the management of large amounts of dependencies of different kinds.

We outline and discuss the differences in the detection results obtained by our tool and other ones on the analysis of one system. The results have been manually checked by three evaluators (PostDoc, PhD and MsC student). For two AS we have also proposed a kind of Filter to remove false positive instances. The main contributions of this work are related to the introduction of a new tool for AS detection that exploits a new technique respect to previous approaches and the introduction of Filters to control the size and relevance of the results. The detection of these AS is the first step towards the detection of new ones by considering also the development history of a system.

The paper is organized as follows: in Sec. II we briefly introduce some related work on AS detection tools; in Sec. III we describe the main components of our tool and the detection algorithms of the three supported AS; in Sec. IV we show the detection results on seven systems, and finally in Sec. V we conclude and outline some future developments.

II. Related Work

Many tools have been developed for code smells detection, and most of them exploit metrics-based detection rules. Few tools are available for architectural smells detection. In particular, we found the following ones.

inFusion [6] supports the detection of both code smells and architectural smells. The tool detects three AS at package level through a combination of some dependency metrics.

Hotspot Detector [7], [8] detects five architectural smells, called Hotspot Patterns, four patterns defined at file level and one at package level. The detector takes as input several files produced by another tool, called Titan [9]. In particular, Hotspot Detector takes as input *1)* a file that contains structural dependencies among files, *2)* another with the evolutionary

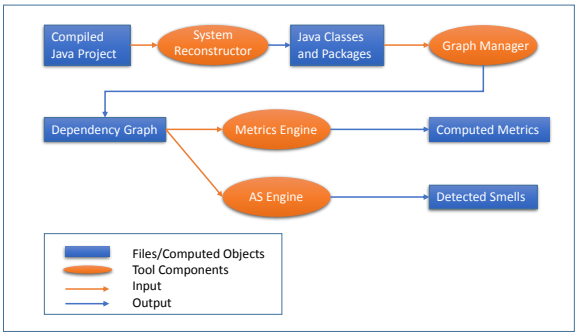

Fig. 1. Tool Workflow

coupling information of files and *3)* a file containing the clustering information of the files. Moreover, Titan uses dependency information provided by the Understand[1] reverse engineering tool.

Other tools will be available to be used in the next future, e.g., SCOOP [2], and another one from Garcia et al. [10].

In this work, we describe the detection of three architectural smells, where two of them are detected also by the above tools (one by Infusion and one by Hotspot detector). We outline that inFusion and its evolution in Aireviewer[2] is a commercial tool and Hotspot Detector is not yet publicly available. Our tool is available at http://essere.disco.unimib.it/wiki/arcan. The detection rules that we exploit are different, as described in the following section.

III. The Arcan Tool

A. Tool Description

We have developed the Arcan tool, with the aim to detect and report architectural anomalies found in Java projects. In particular it focuses on the recognition of architectural smells whose generation was caused by instability issues, where for software instability we mean the incapability to make changes without impacting all the system or a large part of it. To accomplish its aim, the tool computes the metrics proposed by Martin [11] and exploits them during the analysis.

The tool's workflow, as shown in Figure 1, proceeds in four steps:

1) Bytecode Reading: The component[3] named "System Reconstructor" reads files from a compiled Java project; To accomplish its aim, it makes use of the Apache Byte Code Engineering Library (BCEL)[4]: the gathered information is stored in "JavaClass" objects, which represent the structure and content of compiled Java files.

2) Analysis and Graph Generation: From the JavaClass objects, it is then possible to build the dependencies graph. The graph's nodes correspond to Java classes and packages and the edges to dependencies. The component in charge for this step is "Graph Manager", which contains the classes needed to build the graph, initializes the database and writes the graph in it.

3) Architectural Smells Detection: "Metrics Engine" and "AS Engine" components handle the architectural smells recognition. "Metrics Engine" calculates all Martin metrics [11] on which the detection is based. Moreover it computes some cohesion and coupling metrics at class level, e.g., Fan In, Fan Out, CBO [12] and LCOM [13]. The detection algorithms query the graph and apply Filters to obtain the elements involved in smells.

4) Output: The smells detected are written in a series of csv files, showing the architectural flaws at different granularities (class or package).

B. The Dependency Graph

The tool is based on a direct graph, generated through the Apache Tinkerpop[5] graph computing framework and stored in a Neo4j[6] graph database. Since Tinkerpop allows to append some "properties" to both nodes and edges, it was possible to store further information:

- node id, a unique number to identify a node;
- node name, the class or package name;
- node type, depending on whether the class/package is internal to the project under analysis or retrieved from external libraries;
- node modifier, a class property indicating whether a class is abstract or interface.

The edges between nodes are of different kinds:

- dependency between classes, when a class invokes methods belonging to another class;
- hierarchy dependency, directed from "child" classes to their "mother" classes;
- interface dependency, directed from an implementation to an interface;
- afferent dependency, directed from a class outside a package, that depends on a class inside it, to the package itself [11];
- efferent dependency, directed from a class outside a package, which is depended by a class inside it, to the package itself [11];
- membership to package, representing the relationship between packages and the classes belonging to them;
- dependency between packages.

C. The Architectural Smells Detector

The tool provides a detector which runs three different algorithms for smell identification, one per architectural smell. In particular, we have considered the following three architectural smells:

- **Unstable Dependency (UD)**: describes a subsystem (component) that depends on other subsystems that are

[1]https://scitools.com/

[2]http://www.aireviewer.com

[3]In this paper "component" means a package with a specific responsibility in the tool.

[4]commons.apache.org/proper/commons-bcel, Apache BCEL 6.0

[5]http://tinkerpop.apache.org/, Apache Tinkerpop 3.1.1-incubating

[6]http://neo4j.com/, Neo4j 2.3.2

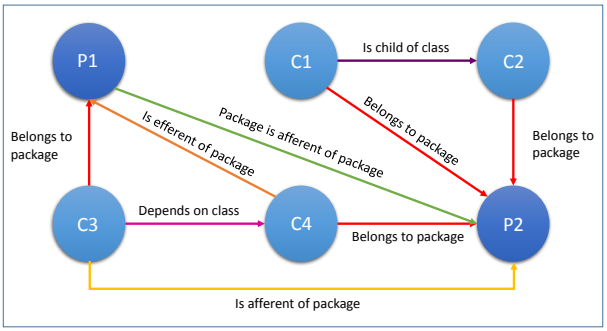

Fig. 2. Dependency Graph Example

less stable than itself. Subsystems affected by this may cause a ripple effect of changes in the system [14].

- **Hub-Like Dependency (HL)**: this smell occurs when an abstraction has (outgoing and ingoing) dependencies with a large number of other abstractions [15].
- **Cyclic Dependency (CD)**: refers to a subsystem (component) that is involved in a chain of relations that break the desirable acyclic nature of a subsystems dependency structure. The subsystems involved in a dependency cycle can be hardly released, maintained or reused in isolation [14].

The algorithms for their detection are described below:

1) Unstable Dependency: The steps to identify this smell are three:

- obtaining from the graph all the dependencies between packages;
- computing the Instability metric for every package of the system;
- for every package, checking if it is afferent of a less stable package.

The results of the detection will be a pool of packages affected by the smell. A further verification can be made by checking the ratio of "bad" dependencies (where "bad" means a dependency which points to a less stable package) to total dependencies. As the result gets closer to 1, the smell can be considered stronger. We use this value as a kind of Filter for the detected smells.

2) Hub-Like Dependency: The steps to identify this smell are the following:

- finding all vertices of type class;
- for all of them, calculating the ingoing and outgoing dependencies;
- calculating the median of the number of ingoing and outgoing dependencies of all the classes of the system;
- checking if the number of ingoing and outgoing dependencies of a class is respectively greater than the ingoing median and outgoing one;
- checking if the difference between ingoing and outgoing dependencies is less than a quarter of the total number of dependencies of the class; if so, the class could be a hub.

The last step defines that the difference between ingoing and outgoing dependencies must be small to consider dependencies as "balanced". After exploring different settings, in the algorithm this quantity was set at $1/4$ of the sum of ingoing and outgoing dependencies.

3) Cyclic Dependency: The tool provides Cyclic Dependency detection for two different granularities: cycles among classes and cycles among packages. To accomplish the detection, the tool relies on a depth first search (DFS) algorithm [16] through the following steps:

1) Extracting the subgraph relative to the requested granularity level (class or package);
2) Launching the DFS algorithm on the subgraph.

TABLE I
ANALYZED PROJECTS

Project	Version	Lines of Code (LOC)	Num of Packages (NOP)	Num of Classes (NOC)
Derby	10.9.1.0	651118	217	3010
Jedit	4.3.2	109515	38	1017
Junit	4.10	6580	28	171
Maven	3.0.5	65685	143	837
Quartz	1.8.6	24522	39	455
Spring	3.0.5	329358	598	4615
Struts	2.2.1	143196	258	2231

IV. Detection Results

The results were retrieved from the analysis on seven open source projects, listed in Table I. After every analysis the tool writes the results in six csv files. The execution of both Unstable Dependency (UD) and Hub-Like (HL) smell detectors generates one file, while Cyclic Dependency (CD) outputs two files. The last files contained in the results pool are the one containing the computed Martin metrics [11] for packages and the one reporting class metrics.

The data contained in files are obtained simply applying the algorithms derived from the definitions of the three architectural smells. Moreover, we decided to apply some Filters to better understand the smells and avoid false positives instances. A Filter is a refinement criterion whose need emerged from the manual analysis of the results: two examples of Filters will be introduced in the next sections. While we checked the extracted results for correctness, we cannot provide precision or recall estimations, because there is no reference data or established evaluation criteria in the literature to be applied. Hence, the aim of the comparison of our results with those of other two tools was only to better check and improve the results of Arcan, that lead to the introducton of Filters. We do not aim to validate the results of the tools in terms of accuracy.

A. Unstable Dependency Results

The output file of this smell consists in a table showing the packages affected by the smell, the packages which cause

TABLE II
UNSTABLE DEPENDENCY RESULTS

Project	Affected Packages Arcan	inFusion
Derby	35	5
Jedit	14	2
Junit	11	0
Maven	69	1
Quartz	8	1
Spring	107	5
Struts	43	7

TABLE III
QUARTZ UNSTABLE DEPENDENCY RESULTS

Package	Arcan	inFusion	Bad dep.%	Filtered
org.quartz.core	yes	no	<30%	no
org.quartz.utils.counter.sampled	yes	no	35%	yes
org.quartz.ee.jta	yes	no	32%	yes
org.quartz.impl	yes	no	<30%	no
org.quartz.utils	yes	yes	100%	yes
org.quartz.impl.jdbcjobstore.oracle	yes	no	<30%	no
org.quartz.utils.counter	yes	no	83%	yes
org.quartz	yes	no	60%	yes

the smell and their respective Instability. Table II displays the unfiltered results of Unstable Dependency's detection on the seven projects, starting from the definition of the smell given before.

Table III instead shows the results retrieved from the analysis of Apache Quartz, which were manually analyzed and inspired the proposal of a Filter. We chose Quartz as an example of medium size project. The first column contains the packages which Arcan considers affected by the Unstable Dependency smell; the second and the third columns compare the detector's results with the InFusion tool's ones; the fourth column shows the results of the application of the Filter described in Section III-C, using a threshold ("bad" dependency ratio) of 30%; the last column shows the new results after the filtering process. Arcan results agree with InFusion's ones on a single package, $org.quartz.utils$, the only one with every dependency compromised. In this example, the Filter threshold was set at 30%, since it highlights the largest share of correct UD instances, according to the manual validation we performed. We also explored different thresholds for the Filters. Figure 3 shows the percentage of packages detected as UD in all the systems, when varying the threshold from 10% to 100%. As we can see, the number of detected packages is almost stable with thresholds $\geq$ 50%.

B. Hub-Like Results

The detection of this smell produces a file containing the list of the detected hub classes. The results retrieved from the analyzed projects are displayed in Table IV. For this smell,

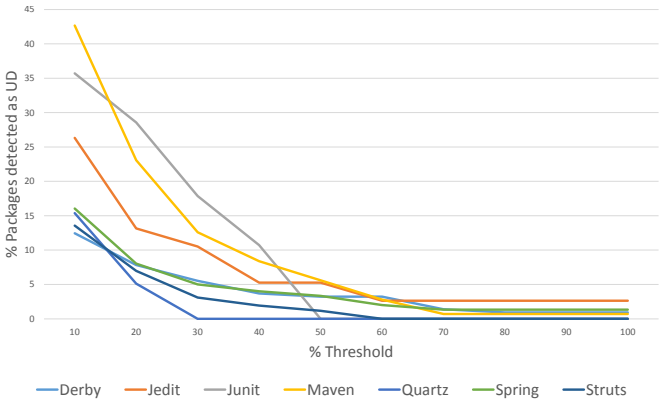

Fig. 3. Filtered Unstable Dependency Results

TABLE IV
HUB-LIKE RESULTS

Derby	Jedit	Junit	Maven	Quartz	Spring	Struts
1	7	1	3	0	2	1

we cannot provide a comparison with other tools, since we are not aware of any tool detecting it.

No filters are needed for this smell. In fact the detection algorithm not only considers classes with a large amount of dependencies in general, but also classes with a balanced number of ingoing and outgoing ones; these are the two characteristics which make them hubs. See Figure 4 for a graphical example of hub, obtained from the graph DB.

While performing our study on the Hub-Like architectural smell, we realized that there are two possible interpretations of this smell, depending on whether all classes with a reference from/to the hub are internal to the project or not. In the first case, the hub would be as defined until now, as an architectural issue. In the second case, for instance given an hub with dependencies equally divided in internal ingoing and external outgoing ones, the hub could be a feature of the architecture instead of an issue. This because the hub class could have been chosen as a controlled exit point, to logically divide the internal project from the external libraries. We do not exclude a third case where an hub with a mixture of internal and external ingoing/outgoing dependencies could be a problem because of its lack of architectural logic.

C. Cyclic Dependency Results

To better manage the results of Cyclic Dependency (CD) detection, two different files offer two points of view of this smell. The first one contains a table with cycles as rows and class/packages as columns: in this way the focus is on the cycles and the elements (packages/classes) which make it up. The second file consists in a matrix with the same elements on rows and columns and with cells filled with counters. These counters refer to the number of times a couple of elements were found in the same cycle. For instance, if two packages are present together in three different cycles, the cell

Fig. 4. Example of Junit Hub-Like class

TABLE V
CYCLIC DEPENDENCY RESULTS

Project	Couples	Couples non-dup	Hotspot
Derby	5520	736	92
Jedit	552	131	42
Junit	110	70	22
Maven	4036	749	32
Spring	0	0	4
Struts	2160	558	54

corresponding to the respective row and column will contain the number 3.

The tool detects cycles relying on the DFS algorithm (see Section III-C), retrieving cycles of different sizes, where often bigger ones contain smaller ones. A proposal of Filter could be to consider only the bigger ones and discard the others, to refine the results and obtain clearer information. Detecting cycles having size ≥ 2 implies that more (and less obvious) cycles will be detected than considering size $= 2$. Table V contains the number of couples of packages found in cycles retrieved from the second output (the one in matrix form) of the CD algorithm: column "couples" contains all couples that were found together in the same cycle, and considers their repetition caused by their presence in more than one cycle; column "couples non-dup" shows the number of couples in cycles too, but counting each one only once. The last column reports Hotspot Detector [7], [8] results, which considers only cycles having size $= 2$, i.e., couples. The differences in the tools' results are justified by their choice of the size of the cycle to detect.

V. CONCLUSION

In this paper, we introduced a new tool for AS detection. We showed how it can detect three different architectural smells and compute Martin's metrics. We also proposed some Filters to better understand the results obtained from the detection and remove false positive instances. The tool was tested on seven open source Java projects, and for two AS we compare our results with those obtained by other two tools able to detect the smells. The tool relies on graphs to represent the extracted information. The application of graph databases makes the graph reusable for further analyses, including the experimentation of new algorithms, and the implementation of new detectors. The applied graph computing technology allows exploiting different graph database backends, while relying with a flexible API. This is also a recent trend in the literature [17]. Further developments could be done to detect other architectural smells. In particular, we are interested in detecting architectural smells using development history. We are studying techniques to reveal hidden dependencies among unrelated files changed frequently together and to detect "leading" files which tend to change with other ones always at the same time: these files are portions of the system which are hard to maintain and are indicators of poor design. Furthermore, we are studying the impact of these architectural smells on system quality through the development history.

Finally, we would like to define and detect through Arcan tool a new technical debt index more focused on the quantification of architectural erosion.

REFERENCES

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

[2] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, "Supporting the identification of architecturally-relevant code anomalies," in *Proc. 28th Intl Conf. Softw. Maintenance (ICSM 2012)*. Trento, Italy: IEEE, Sep. 2012, pp. 662–665.

[3] M. Lippert and S. Roock, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, Apr. 2006.

[4] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms," in *Proc. 15th European Conf. Softw. Maintenance and Reeng. (CSMR 2012)*. Hungary: IEEE, Mar. 2012, pp. 277–286.

[5] R. C. Martin, "Object oriented design quality metrics: An analysis of dependencies," *ROAD*, vol. 2, no. 3, Sept–Oct 1995.

[6] Intooitus, "inFusion," 2015, http://www.intooitus.com/products/infusion.

[7] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *Proc. 12th Work. Conf. Soft. Arch. (WICSA 2015)*, Montreal, Canada, 2015.

[8] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A case study in locating the architectural roots of technical debt," in *Proc. 37th Intl Conf. Softw. Eng. (ICSE 2015)*, vol. 2. Florence, Italy: IEEE, May 2015, pp. 179–188.

[9] L. Xiao, Y. Cai, and R. Kazman, "Titan: A toolset that connects software architecture with quality analysis," in *Proc. 22nd Int. Symp. Found. of Soft. Eng. (FSE 2014)*. China: ACM, Nov. 2014, pp. 763–766.

[10] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *Proc. 26th Intl Conf. Automated Softw. Eng. (ASE 2011)*, Nov. 2011, pp. 552–555.

[11] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[12] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[13] B. Henderson-Sellers, *Object-Oriented Metrics, measures of complexity*. Prentice Hall, 1996.

[14] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM J. Research Develop.*, vol. 56, no. 5, p. 9, 2012.

[15] W. Tracz, "Refactoring for software design smells: Managing technical debt by girish suryanarayana, ganesh samarthyam, and tushar sharma," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 6, p. 36, 2015.

[16] R. Sedgewick and K. Wayne, *Algorithms 4thEd*. Addison-Wesley, 2011.

[17] R. Dąbrowski, K. Stencel, and G. Timoszuk, "Software is a directed multigraph," in *Proc. 5th European Conf. Softw. Arch. (ECSA 2011)*. Essen, Germany: Springer, Sep. 2011, pp. 360–369.