Arcan: a Tool for Architectural Smells Detection

Francesca Arcelli Fontana*, Ilaria Pigazzini†, Riccardo Roveda*, Damian Tamburri‡,
Marco Zanoni*, Elisabetta Di Nitto‡
Università degli Studi di Milano - Bicocca, Milan, Italy
Email: {arcelli,riccardo.roveda,marco.zanoni}@disco.unimib.it*, i.pigazzini@campus.unimib.it†
Politecnico di Milano Dipartimento di Elettronica, Informatica e Bioingegneria, Milan, Italy
Email: {damianandrew.tamburri,elisabetta.dinitto}@polimi.it‡

***Abstract*—Code smells are sub-optimal coding circumstances such as blob classes or spaghetti code - they have received much attention and tooling in recent software engineering research. 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. This paper describes an open-source tool called Arcan developed for the detection of architectural smells through an evaluation of several architecture dependency issues. 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 the scope of this paper, we focus on the evaluation of Arcan results carried out with real-life software developers to check if the architectural smells detected by Arcan are really perceived as problems and to get an overall usefulness evaluation of the tool.**

I. Introduction

It is an established fact that good software architecture and design lead to better evolvability, maintainability, availability, security, software cost reduction and more [1]. Conversely, when that architecture and design process are compromised by poor or hasted design choices, the architecture is often subject to different architectural problems or anomalies, that can lead to software faults, failures or quality downfalls such as a progressive architecture erosion [2], [3]. A category of these anomalies is represented by architectural smells, that are caused by a violation of recognized design principles with a negative impact on internal system quality [4].

To aid the detection and removal of architecture smells (AS), this paper introduces Arcan, a static-analysis software useful to support software developers and designers during the development, maintenance and evolution of Java applications. Arcan is able to detect 3 architectural smells (i.e., Cycle Dependency, Unstable Dependency and Hub-Like Dependency). We focused our attention on AS related to dependency issues; we will consider other AS in the next future. The design of our tool relies on recent advances in graph database technology and graph computing: once a Java project has been analyzed by Arcan, a new graph database is created containing the structural dependencies of the system. Thanks to graph computing and connected big data processing technology [5], it is then possible to run detection algorithms on this graph to extract information about the analyzed project (package/class metrics, architectural issues).

In a previous work [6], we outlined the detection algorithms hardcoded within Arcan. The original contribution of this paper is its focus on: (a) the tool's architecture and inner workings; (b) on the improvements of its detection strategies for the Cycle Dependency smell, and (c) on the manual validation of the detection results of Arcan done by real-life software developers. The latter exercise enabled us to evaluate whether the detected architectural smells are indeed perceived as problems by the software developers responsible for them. By means of this evaluation exercise, we were able to provide a first evaluation on how the tool works and on its results in terms of precision and recall. According to the feedback of this validation, we outline and discuss future directions for research and extensions of the tool. More in particular, we learned the need for a Severity Index to identify the most critical smells to be analyzed and removed first. This Index would help developers to identify and estimate refactoring needs and their rough cost.

II. Related Work

As previously stated, many tools have been developed for code smells detection but only few tools are currently available for architectural smells detection. The following briefly reports on some of them. First, the commercial tool inFusion[1] and its evolution in AiReviewer[2] support the detection of both code smells and some design or architectural smells. Another commercial tool is Designite[3] that detects several design smells in C# projects. The Hotspot Detector [7] tool detects five architectural smells, called Hotspot Patterns, four patterns defined at file level and one at package level. Other tool prototypes have been proposed, e.g., SCOOP [8], and one from Garcia et al. [9]. We outline that AiReviewer and Designite are commercial tools and according to our knowledge the other tools are not yet publicly available. Moreover, there are different commercial tools as for example Sotograph, Sonargraph, Structure 101 and Cast which are able to detect different kinds of architectural violations, as dependency cycles. Our tool (available at http://essere.disco.unimib.it/wiki/arcan), by analyzing compiled Java files, detects three AS. We compute the Cyclic Dependency AS among classes and packages, and we detect it according to different shapes. We exploit different

[1]Intooitus, http://www.intooitus.com/products/infusion
[2]http://www.aireviewer.com
[3]http://www.designite-tools.com

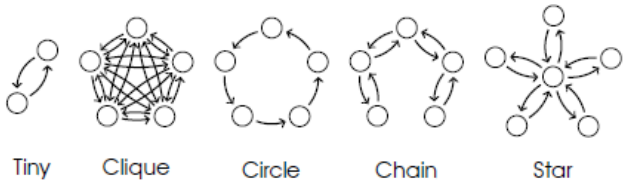

Fig. 1. Cycles shapes [11]

detection techniques with respect to the previous approaches, that, according to our knowledge, have not been applied for architectural smells detection before. It is possible to inspect the results through graphs, more useful respect to other views, allowing to better identify refactoring opportunities for the architectural smells.

III. Architectural Smells

As previously introduced, Arcan detects 3 architectural smells described below, that may be causing instability issues, where instability refers to the predisposition of objects to change [10]. The instability is computed as the ratio between efferent dependencies over the total number of dependencies, where the efferent dependencies are the number of classes inside the package that depend upon classes outside the package. In the following for subsystem (component) we mean a set of packages and classes which identifies an independent unit of the system responsible for a certain functionality.

1) Cyclic Dependency (CD): (detected on classes and packages) refers to a subsystem (component) that is involved in a chain of relations that break the desirable acyclic nature of a subsystem's dependency structure. The subsystems involved in a dependency cycle can be hard to release, maintain or reuse in isolation. For what concerns this smell, we found useful to refine the detected cycles according to their shape [11] (shown in Figure 1). We established rules to identify each shape. Some of them are formulas which set the relationship between the number of nodes and edges; others are constraints at graph level, i.e., patterns that nodes and edges have to follow to make up a certain kind of shape.

2) Unstable Dependency (UD): (detected on packages) describes a subsystem (component) that depends on other subsystems that are less stable than itself, according to the Instability metric value [10]. This may cause a ripple effect of changes in the system.

For this smell, we 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, it was interesting to examine the dependencies which actually cause the smell. Considering "Bad Dependency" a dependency that points to a less stable package, we proposed a formula to establish the "Degree of Unstable Dependency" (DoUD):

$$DoUD = \frac{BadDependencies}{TotalDependencies} \quad (1)$$

A package with a small number of bad dependencies may not be a smell, and this formula helps to filter misleading

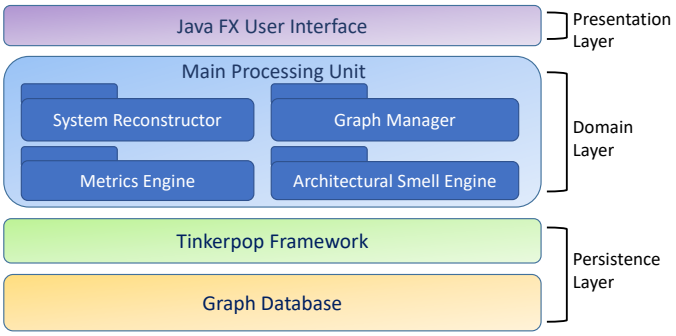

Fig. 2. Arcan Architecture

results. The Degree of Unstable Dependency under which a package is no more a smell can be defined as a *threshold* of the filter. The filter threshold establishing the minimum level needed to consider a package affected by the smell (DoUD) was set at 30%, since this value highlights the largest share of correct UD instances, according to a manual validation we performed while exploring different thresholds over several projects. We are working on the definition of new methods for the identification of the right thresholds and to adapt them to the developers' needs. In particular, we aim to discover the right thresholds through machine learning techniques.

3) Hub-Like Dependency (HL): (detected on classes) this smell arises when an abstraction has (outgoing and ingoing) dependencies with a large number of other abstractions [12]. For this smell, we figured out which are the conditions where a Hub-Like could be a false positive instance. First, HL are highly used classes of the system. Assuming we don't know if those classes are used from other systems (since Arcan analyzes one system at a time), we conjectured that if the class uses external classes of the system, e.g., classes of the package `java.util.*`, and these are the majority of the total outgoing dependencies related to a system library, then it should **not** be considered a Hub-Like class. In fact, 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 main functionalities of the system exhibit the opposite pattern.

IV. Arcan: Architecture and Inner Workings

The Arcan architectural core consists of four components (see Figure 2), structured in a layered architecture style: a user interface built with Java FX, a main processing unit with all the logic components and a persistence layer consisting of a graph database, accessed through a graph computing framework. The tool is written in Java 8. To interface with the graph database, we decided to use Apache Tinkerpop[4] for two reasons: to easily build and access the dependency graph which represents the analyzed project and to allow the exploitation of different graph database backends. This means

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

that all the graph elements, e.g., nodes, are Tinkerpop elements. Every read or write operation that regards the database is filtered by the framework. Hence, the queries are written in Gremlin-Java, the variant of the Gremlin[5] query language that allowed us to write graph traversals within the native Java environment. Tinkerpop deals with the translation of the dependency graph into specific backend's graphs and hides the underneath database. In the current version of Arcan, the graph generated through Tinkerpop can be stored *1)* in-memory or *2)* using a Neo4j[6] graph database; other backends can be added in the future. We chose Neo4j because it offers an intuitive graphic interface which allows the exploration (using Cypher[7] queries) and visualization of the dependency graph built by Arcan. The graph can be browsed to understand the structure of the system and different algorithms can be applied on it to extrapolate more detailed information. After the execution of these algorithms, new nodes and edges are added to the graph as "smell" nodes, which indicate the presence of an architectural smell in the system.

In the following we describe the workflow Arcan applies for the detection of the three architetcural smells.

1) The *System Reconstructor* reads the compiled Java files, which can be submitted as a folder of `.class` files or a folder of `.jar` files. Arcan only retrieves classes and packages which are included in the input, without extending the analysis to external components. Hence, to make Arcan analyze a complete project, it needs to have every component as input. The information contained in the compiled file are extracted thanks to the Apache Byte Code Engineering Library (BCEL[8]). This library offers a class named `JavaClass` to represent the data structures, constant pool, fields, methods and commands contained in a typical Java `.class` file.

2) The *Graph Manager* is the component dedicated to building the dependency graph. From the `JavaClass` object extracted by the *System Reconstructor*, it is possible to know the system classes, packages and references which link to them. These elements are all included in the dependency graph through Tinkerpop. This component also manages the initialization of the database and writes the dependency graph in it.

3) The *Metrics Engine* computes R. Martin's metrics [10], used in the detection of the architectural smells. Moreover this engine is entrusted with computing typical cohesion and coupling metrics at the class level, such as Fan In, Fan Out, CBO and LCOM [13]. To compute these metrics, this component accesses the dependency graph and the results are stored as attributes in the nodes representing the classes or packages that the metrics refer to.

4) The *Architectural Smell Engine* contains the logic for both architectural smell detection and filtering of false positive instances. Every detection algorithm extracts a subgraph from the whole dependency graph and works on it depending on the elements which can be affected by the anomaly: classes or packages. When a smell is detected, a new node of type "smell" (called "supernode") is created and linked to the nodes involved in the detection. This makes easier to filter the results in a second step, when necessary.

[5]http://tinkerpop.apache.org/gremlin.html

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

[7]https://neo4j.com/developer/cypher/

[8]http://commons.apache.org/proper/commons-bcel, Apache BCEL 6.0

V. Validation of Arcan Results

Although in a previous work we experimented Arcan on several projects [6], a more careful qualitative evaluation of Arcan results performed by external tool developers other than ourselves was never previously performed. In the following, we report on our first attempts at this kind of validation on the following two projects:

1) DICER[9],[10]: a continuous architecting tool for data-intensive applications (DIAs) that allows to quickly put together a model of a data-intensive application using known DIA middleware such as Apache Spark, Apache Hadoop MapReduce and Apache Storm. *2)* Tower4Clouds[11]: a flexible, self-adaptable and auto-configurable monitoring infrastructure engineered for multi-cloud applications. Tower includes multiple data-collectors that allow monitoring, collecting and sifting from multiple data sources by means of a rule-based approach.

The evaluation was carried out by direct observation of Arcan results. In particular, for each tool under study, 3 professional software designers experienced with the tool discussed (separately) the Arcan evaluation-sheets line-by-line, quickly checking the code and/or via available tool documentation and deliverables to confirm/refute Arcan findings. These practitioners reported on: *(a)* whether Arcan actually uncovered known or unknown architecture issues; *(b)* whether the issues were actually issues; *(c)* whether refactoring was needed or planned following Arcan results. Reports were captured using in-line comments directly on Arcan result plots – this data is freely available online[12] to encourage verifiability.

TABLE I
Analyzed Projects

Projects	DICER	Tower4Clouds
Version	0.1.0	0.3.1
Packages(NOP)	549	373
Classes(NOC)	13204	8820
Analyzed Component	it.polimi.dice.dicer	it.polimi
Packages(NOP)	9	7
Classes(NOC)	36	111

Size metrics for the projects are shown in Table I. The total number of AS in the projects, and the evaluation of Arcan detection performances is reported Table II. We report standard Information Retrieval performance metrics, i.e., confusion matrix elements and derivatives, like precision and recall.

We observed a precision of 100%, since Arcan found only correct instances of architectural smells but developers reported 5 more architectural smells which are False Negatives.

[9]https://github.com/dice-project/DICER

[10]Some of the authors' work is partially supported by the European Commission grant no.644869 (H2020 - Call 1), DICE.

[11]http://deib-polimi.github.io/tower4clouds/docs/overview.html

[12]http://tinyurl.com/zpquemg

TABLE II
ARCHITECTURAL SMELLS IN THE ANALYZED COMPONENT

	DICER	Tower4Clouds
Total Architectural Smells	5	9
True Positive	3	6
False Positive	0	0
False Negative	2	3
True Negative	0	0
Precision(%)	100	100
Recall(%)	60	66
F-measure(%)	75	79,52

TABLE III
DETECTED ARCHITECTURAL SMELLS BY ARCAN

	DICER	Tower4Clouds
Cyclic Dependency (class)	636	439
Cyclic Dependency (package)	83	38
Unstable Dependency	305	123
Hub Like Dependency	1	3
Totals	1025	603

False Negatives were related to external components out of the scope of the analysis of the tool.

As we can see from Table I, the developers focused their attention on a component of the projects, and not on the AS found in the entire projects: this is caused by the high number of the detected AS. In particular, for the CD smell Arcan found 636 occurrences in DICER and 439 in Tower4Clouds (see Table III). As a consequence, using followups and feedback from the evaluation we decided to define a Severity Index for the CD smell; the purpose of the index is to assist in the identification of the most critical smells to be analyzed and then removed first. The *Severity Index* is defined as follows: it counts the number of vertices involved in a cycle and the weight (number of occurrences) of every edge which forms the cycle, then orders the instances of Cyclic Dependency descending by the number of vertices in the cycle and the maximum number of times the cycle occurs.

Moreover, the results of Arcan on the analysis of 8 projects of the Qualitas Corpus, have been manually evaluated by three Master students with a high background on the subject. They found the tool installation very easy and the interface intuitive. From this evaluation we got important hints on how to improve Arcan results to remove possible false positive instances: (a) the detected false positives for Hub Like Dependency smell reflect abstract classes, interfaces and classes which implement the Singleton design pattern; (b) the Cyclic Dependency smell false positives reflect classes which implement Factory Method design pattern and nested (hidden) classes. In the future we shall use these insights to offer a more refined version of our tool to better assist its usage in AS refactoring scenarios.

VI. CONCLUSION

In this paper, we introduced the Arcan tool for architectural smells detection. The aim of Arcan is to support the automatic analysis of software architecture through a graph representation of data, providing support during the software development and maintenance processes. Architectural smells dig up complexity in the dependency graph as a bottleneck referring to Hub Like AS, as the violation of the acyclic dependency principle regarding Cyclic Dependency AS and as the violation of stable dependency principle with respect to Unstable Dependency AS. We offer an experimentation on 2 projects developed in collaboration with industry - our goal was to validate Arcan results and get feedback by the developers/maintainers of analysed projects.

In the future, we plan to investigate how architectural smells evolve in the history of a project and how they affect the overall technical debt. Moreover, we aim to define a new technical debt index more focused on architectural issues [14]. We currently detect only three AS, we are interested to detect other architectural smells, also not strictly related to dependency issues and include a new component devoted to the suggestion of the refactoring steps to remove the architectural smells, since their refactoring could be a critical task. In this context, we aim also to identify the most critical ones exploiting our Severity Index in order to provide some refactoring needs estimations and to define other possible filters to remove false positive instances [15]. Finally, we aim to study the impact that the different CD shapes could have on architetcure erosion.

REFERENCES

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison Wesley, 1998.

[2] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *Journal of Systems and Software*, vol. 61, no. 2, pp. 105–119, 2002.

[3] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *J. Syst. and Software*, vol. 85, no. 1, 2012.

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

[5] N.-L. Tran, S. Skhiri, A. Lesuisse, and E. Zimanyi, "Arom: Processing big data with data flow graphs and functional programming." in *Cloud-Com*. IEEE Computer Society, 2012, pp. 875–882.

[6] F. Arcelli Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic detection of instability architectural smells," in *Proc. of the 32nd Intern. Conf. Soft. Maint. and Evol. (ICSME 2016)*. USA: IEEE, Oct. 2016.

[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] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, "Supporting the identification of architecturally-relevant code anomalies," in *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012)*, Sept 2012, pp. 662–665.

[9] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *Proc. 26th IEEE/ACM Intern.Conf.Aut.Soft.Engin.(ASE 2011)*, USA, Nov. 2011.

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

[11] H. A. Al-Mutawa, J. Dietrich, S. Marsland, and C. McCartin, "On the shape of circular dependencies in java programs," in *Proc. 23rd Austr.Soft.Engin.Conf. (ASWEC 2014)*. Australia: IEEE, Apr. 2014.

[12] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells*, 1st ed. Morgan Kaufmann, 2015.

[13] 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.

[14] F. Arcelli Fontana, R. Roveda, and M. Zanoni, "Technical debt indexes provided by tools: a preliminary discussion," in *Proc. of the 8th Intern. Workshop on Managing Technical Debt (MTD 2016)*, 2016.

[15] F. Arcelli Fontana, V. Ferme, and M. Zanoni, "Poster: Filtering code smells detection results," in *Proc. 37th International Conference on Software Engineering (ICSE 2015)*. Florence, Italy: IEEE, May 2015.