Architectural Smells Detected by Tools: a Catalogue Proposal

Umberto Azadi
Università degli Studi di Milano - Bicocca
Milan, Italy
u.azadi@campus.unimib.it

Francesca Arcelli Fontana
Università degli Studi di Milano - Bicocca
Milan, Italy
arcelli@disco.unimib.it

Davide Taibi
Tampere University
Tampere, Finland
davide.taibi@tut.fi

***Abstract*—Architectural smells can negatively impact on different software qualities and can represent a relevant source of architectural debt. Several architectural smells have been defined by different researchers. Moreover, both academia and industry proposed several tools for software quality analysis, but it is not always clear to understand which tools provide also support for architectural smells detection and if the tools developed for this specific purpose are effectively available or not. In this paper we propose a catalogue of architectural smells for which, at least one tool able to detect the smell exists. We outline the main differences in the detection techniques exploited by the tools and we propose a classification of these architectural smells according to the violation of three design principles.**

***Index Terms*—Architectural Smells, Architectural Smells Catalogue, Architectural Smells Detection, Architectural Debt**

I. Introduction

Architectural issues, as outlined by Ernst et al. [1], are one of the greatest sources of technical debt. Hence, it is important to understand how to identify and manage them, to avoid and reduce technical debt accumulation. Architectural smells (AS) can be seen as the code smells [2] counterpart at the architecture level. ASs represent the violation of design principles or decisions that impact on internal software qualities [3] with large negative effects on maintenance and evolution costs [4]. They represent a source of architectural debt [5], important to identify, investigate and remove through different refactoring steps.

The interest on ASs is growing during the last years ([5][6][7][8]), and different authors provided different definitions of ASs and in some cases they also provided the tool to detect the AS. According to our previous experience in using and comparing different tools for software maintenance [9], software quality assessment [10][11], code and architectural smells detection [12][13], we focus here our attention on the available tools that can be used to automatically detect ASs. Moreover, the application of different detection mechanisms for the same AS could lead to different results. Therefore, we are also interested to compare the detection techniques adopted by the tools.

Hence, the aim of this paper is to propose a catalogue of ASs that can be detected by tools and compare the detection techniques adopted.

In order to investigate our goal, we identified a set of tools and in particular we selected a list of nine tools. We did not consider tools that are no longer available on the market for several years.

According to our aim, we collected the following data: *(i)* the list of the ASs detected by the tools, *(ii)* the smell definition on which the tool relies for the smell detection, *(iii)* at least a high-level description or the implementation of the detection strategy/rule used for each AS. Furthermore, these data have been verified through our usage of the tool if available or otherwise as reported in a recent published paper on the tool (see Section III-A).

Each tool could detect a various set of smells, however we decided to focus our attention only on ASs, i.e. poor or rushed design choices which can cause architectural problems or anomalies that can lead to software faults, failures or quality downfalls such as a progressive architecture erosion [13]. We are aware that some code smells proposed by Fowler [2] can have an impact on different architecture issues, but we have not considered here this kind of smells. This definition of AS is also consistent with the definition accordingly to which "a software system's architecture corresponds to the set of principal design decisions made during its development and any subsequent evolution" [14]. It is important to specify that, according to this definition, we will consider as ASs some smells that are sometimes indicated as design smells or anti-patterns, while we do not consider as architectural all the smells that do not respect the definition reported above.

Hence, in this work, we aim to answer the following questions:

- Q1: Which ASs are detected by at least one tool? can we provide a catalogue of these AS?
- Q2: How an AS is detected? which are the main differences in the detection rules adopted by different tools?
- Q3: Which kind of classification/s of AS can be proposed?

The answers to these questions can be useful to *(i)* software developers/maintainers to be able to identify which AS can be detected by a tool, to choose the tool to be used according to their aims or to identify the tool able to detect the largest number of AS and *(ii)* can be useful to the developers of the detection tools to immediately get an overview of the different used detection techniques, to be able to exploit or improve them. According to AS classification, a classification is useful to group ASs according to some features or the impact

they could have on some quality attributes or the violation of some well known design principles. Therefore, we propose a classification based on design principle violation.

In order to answer our questions, we propose a new template to be used in the AS catalogue, where each smell is defined according to different features (described in Section III-B). Among them, we take into particular account the detection techniques exploited by the tools outlining the main differences between them.

This template has been produced by adapting the template proposed by Ganesh et al. [15], in order to make it more suitable for our purpose focused on the AS detection support. The differences and the similarities between the two templates are described in detail in Sections III-B and III-C.

The paper is organised through the following sections: In Section II we report some related works, in Section III we introduce the study that has been accomplished by describing *(i)* how the tools have been selected, *(ii)* the template used to document the ASs in the catalogue and *(iii)* the principles through which the ASs have been classified. Section IV contains the catalogue of the ASs. In Section V, we discuss the proposed ASs classification. Finally, in Section VI we conclude our work and describe some future developments.

II. Related works

Different works have been proposed in the literature on code smells [16], on the comparison of code smell detection techniques or tools i.e [17], on code smell classification [18] and the well known catalogue provided by with the definitions of 23 code smells of Fowler [2]. Less work has been done on AS catalogue and classification.

Garcia et al. [19] provided the first catalogue of AS, describing each AS with the related quality impact and trade-offs and providing a generic schematic view by means of UML diagrams. However, this catalogue contains only four AS and no reference to the detection support is provided.

Le et al. [20] proposed a classification of AS based on four categories: Interface, Change, Dependency and Concern-based smells. They have defined different ASs in these categories and outlined the impact of these smells on different quality attributes (e.g. complexity, reusability, modularity,..).

Lippert and Rook [3] defined and classified different AS at different levels, by considering three categories of issues related to Dependency, Inheritance and Size, with respect to packages, subsystems, and layers. In particular, they defined AS smells in dependency graphs, inheritance hierarchies, packages, subsystems and layers.

Ganesh et al. [21] provided a catalogue of smells with a focus on the so called *design smells*, including code smells [2], antipatters [22] and ASs. We discuss in details their classification and compare it with our classification in Section III-C and Section V.

Mo et al. [23] proposed a list of AS based on Baldwin and Clarks design rule theory [24] and basic software design principles. They summarised these architecture issues into five architecture hotspot patterns defined at file and package levels.

Hence, our work differs from previous works, since we provide a catalogue of only AS for which we know there is at least a tool able to detect them. We outline the main differences in the detection techniques exploited by the different tools and we provide a classification of these AS according to three different categories described in Section III-C.

III. Study Design

A. Selection of the detection tools

According to our experience in developing an AS detection tool [25], and our great interest in this area [6], [13], we hope we reached a good knowledge on the community working on the development of tools able to detect AS.

Hence, according to this knowledge we identified a list of tools reported in Table I which we have considered for this study. In Table I we report other useful information about the tools (supported platform, languages[1], licence and if the tool is currently supported and available). Obviously, we want to underline that the reported tools do not cover the whole panorama of the available tools that can be used to evaluate software architecture quality. We focus here our attention on tools able to detect AS of some kind. Furthermore, we were not able to gather all the required information (the smell definitions and the detection strategies) for some tools, such as CAST [26], which allows to detect cyclic dependency and other architectural violations, or Scoop [27] and Titan [28] tools. Furthermore some tools are no longer available since several years, such as inFusion and SA4J.

Hence, a specific tool has been selected only if we managed to gather a certain amount of information about it from at least one "official" source, i.e. the website of the tool, a published paper, the tools GUI or the tools source code. Moreover, if the tool was available we experimented it by analysing different projects in order to detect the AS. We could obviously have omitted some tools and some ASs in this study, that can be integrated into a next refinement of the catalogue. As already outlined, we experimented some of the tools of Table I for different purposes ([11],[12]).

For this study, we focused our attention on the identification of the AS detected by each tool. An AS has been included in the catalogue only after the tools inspection to check which AS are detected by each tool. We found different cases: AS detected only by a specific tool, AS detected by more tools and sometimes called and detected in different ways and other scenarios described in Section IV.

B. Architectural smells catalogue definition

The template that we define to present each AS is based on the one defined by Ganesh et al. [15]. However there are several differences due to our intended purpose related to the detection support of each AS. Indeed, they clearly aim to create a catalogue where every facet of each AS is taken in consideration and deeply described through different fields which include a Short description, a Long description,

[1] Currently, we found only tools based on object-oriented language analysis

TABLE I
ANALYSED TOOLS

Tool name	Supported platform	Supported languages	Licence	Currently Available	Ref
AI Reviewer	Windows, Linux, Mac OS	C, C++	Commercial	Yes	[29]
ARCADE	Windows, Linux, Mac OS	Java	Free	No	[30][20]
Arcan	Windows, Linux, Mac OS	Java	Free	Yes	[31]
Designite	Windows	C#	Commercial	Yes	[32]
Hotspot Detector[2]	N.A.	Java	N.A.	No	[23]
Massey Architecture Explorer	Windows, Linux, Mac OS	Java	Free	No	[33][34]
Sonargraph	Windows, Linux, Mac OS	Java, C#, C, C++	Commercial	Yes	[35]
STAN	Windows, Linux, Mac OS	Java	Commercial	Yes	[36]
Structure 101	Windows, Linux, Mac OS	Java, .Net, C, C++ and other 10	Commercial	Yes	[37]

Rationale and Example(s) for each smell. Hence, a great part of their template describes the AS or states the problems related to it, and only two fields in their template provide some hints on the detection and refactoring strategies, without delineate any actual algorithm. While in our approach we provide only a brief description of each AS, and we present a comparison between the actually implemented detection strategies used by each tool for each of the detected smell. The reason why we decided to not describe in detail each smell is based on the fact that they have been already introduced, defined and published in the literature.

However, we noticed that different tools used different definitions, therefore the description that has been provided for each AS in the catalogue does not match exactly any of the definition used by the tools, but it is an attempt to incorporate them into a more general description.

The template that we propose in order to present each AS is organised as follows:

- **Name:** the name through which most of the tools refers to that AS. In the absence of a majority, we selected the one that we consider to be the most intuitive and representative;
- **Also known as:** the names through which the tools refer to that AS, that are different from the one specified in the `Name` field. As for example Cyclic Dependency and Tangle smells;
- **Description:** a brief description of the AS, which in most cases will be deliberately very general in order to include all the definitions used by each tool;
- **Variant/s:** ASs that are fundamentally identical, but exhibit a slight variation from the main smell. For example, the variation may include a special case (as in the case of Hub-Like Dependency and Overreliant Class) or a more general form of the AS (as in the case of Hub-Like Dependency and Dense Structure);
- **Violated principle/s:** object-oriented design principles whose violations can lead to this smell. The principles taken into consideration are those defined in Section III-C;
- **Tools:** tools that are able to detect the smell;
- **Detection Comparison:** a comparison between the detection strategies accomplished by each tool. Furthermore, the default thresholds used for the detection are going to be reported when available; it is important to mention that some of the considered tools allow to customise them.

C. Architectural smells classification

We propose a classification of the AS according to the violation of some design principles in relation to each smell. We considered the classification proposed by Ganesh et al. [15], based on four design principles: Modularity, Hierarchy, Abstraction and Encapsulation. As asserted by the authors [21] [15], *the rationale of this classification is that it enables an intuitive understanding of the smell and it allows to get a better idea on how to refactor the AS*. We found this type of classification very useful because in many cases it helps to retrieve the decision that had caused the occurrence of an AS.

We decided to not consider the Abstraction and Encapsulation principles because they are more closely related to the concept of code smells. While, as explained in Section I, we focus our attention on the concept of AS. Moreover, we defined a new principle, called **Healthy Dependency Structure**.

Therefore, the principles that we selected for our classification are the following:

- **Modularity** [15]: Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.
- **Hierarchy** [15]: Hierarchy is a ranking or ordering of abstractions, where an abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provides crisply defined conceptual boundaries, relative to the perspective of the viewer.
- **Healthy Dependency Structure**: The dependency structure of a (sub-)system is considered *unhealthy* when it promotes a chain of changes in the system each time it is modified.

The **Healthy Dependency Structure** principle aims to group some of the already well-known best practices that involve dependencies, which are: *(i)* the desirable acyclic nature of a subsystem's dependency structure [38], *(ii)* the desirable equality of stability between the two elements (classes or packages) that are involved in a dependency relationship [38] and *(iii)* all the principles that aim to prevent the occurrence of the Rigidity [38], which is a symptom of rotting design.

Therefore, it is possible to see how this principle, that we defined for our classification, is related to the best practices and to the symptoms of rotting design that have been introduced by Robert C. Martin [38]. Furthermore, our main objective

[2]Hotspot Detector has been incorporated in a new tool called Archdia (https://www.archdia.net/)

through this principle is to identify a set of ASs that negatively affect (i.e. increase) the change-proneness of the classes.

It is important to underline the difference that occurs between the Healthy Dependency Structure and the Modularity principles, which in practice might often be violated together. An example, useful to highlight the distinction, is the way in which a God Component smell (Section IV.6) might become a Hub-Like class smell (Section IV.2): a God Component implements a significant amount of business logic, that however can be implemented with few public methods (used by the other classes) and a vast majority of private method (useful only to carry out all the concerns). Therefore, the Modularity principle is violated because of the lack of separation of concerns but the dependency structure is healthy because the change in the God Component does not promote changes in other classes. In such a situation, through a poor refactoring process, it is possible to separate the concern into several classes that use or are used by the God Component. Therefore, the problematic class is no longer a God Component (because the concern has been spit), but it will become a Hub-Like class, for all the dependencies that are generated during the refactoring process. Therefore, the dependency structure has become unhealthy because a slight adjustment in the Hub-Like class will generate a ripple effect, while the modularity is at least improved by the refactoring, because the concerns have been separated.

IV. Architectural smells catalogue

In this section, we provide the catalogue of the AS following the template described in Section III-B. A key aspect of this template is the so-called “Violated principles” by the AS that we use for the classification of the smells (see Figure 2).

1. Cyclic Dependency (CD)

- **Also known as:** Tangle [37] [36], Cross-Module Cycle [23], Cross-Package Cycle [23], Cycle of classes [29], Cyclically-dependent Modularization [32].
- **Description:** this smell arises when two or more architecture components depend on each other directly or indirectly.
- **Variants:**
  - *Strong Circular Dependencies Between Packages* [33]: in this case, a custom metrics called Antipattern score [39] is used to select the most severe occurrences of this AS.
  - *Shape detection* [25]: in this case, for each instance of the smell detected, the tool can also highlight its “shape”, i.e. the topology of the cycle. The five shapes that can be already detected by at least one tool are reported in Figure 1.

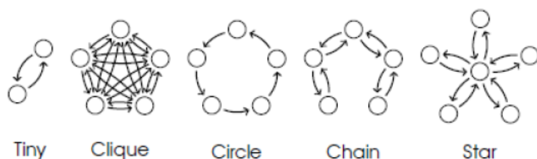

Fig. 1. Cycles shapes [25]

- **Violated principles:**
  - *Healthy Dependency Structure*: the subsystems involved in a dependency cycle can be hard to release and maintain. Furthermore, the presence of this AS implies that the participating classes and packages cannot be deployed and maintained separately.
  - *Modularity*: the presence of this AS implies that there are two pieces of code, that are highly coupled to each other in a direct or indirect way. This situation might suggest that the responsibilities are not separated correctly.
- **Tools:** All the tools that have been taken into consideration in this work detect this smell. However, it is important to mention that there are many other tools that allow to detect only the Cyclic Dependency, due to its relevance and criticality [3][21]. Some examples[3] are: ClassCycle, Dependency Finder, JArchitect, JDepend, Lattix LDM, NDepend, CAST.
- **Detection Comparison:** The tools usually (Designite, Massey Architecture Explorer, Sonargraph, STAN, Structure 101) detect this smell only at the class level, then analyse this smell at package level through the generalisation of the dependency graph at class level obtained using the quite standard rule: if a class *a*, contained in package `A`, depends on class *b*, contained in package `B`, then the package `A` depends on package `B`. This type of analysis is useful for a visual representation because it allows to start with a high level view and then expand the most relevant packages. The detection approach previously mentioned is also used by others tools (Arcan, Hotspot Detector), however, they treat the packages and the classes as distinct elements and perform a two layers analysis. The possibility of considering these two layers individually during the architecture quality assessment can be very useful for many reasons, as for example the prioritisation of the refactoring, in fact the intra-package cycles are usually considered less problematic compared to the inter-package cycles. Finally, AI Reviewer and ARCADE consider this smell only at class level.

2. Hub-Like Dependency (HLD)

- **Also known as:** Hub-like Modularization [32], Link Overload [20].
- **Description:** The Hub-Like Dependency smell occurs when an abstraction or a concrete class has (outgoing and ingoing) dependencies with a large number of other abstractions or concrete classes.
- **Variants:**
  - *Overreliant Class* [29]: this smell refers mostly only to the outcoming dependencies. This AS is often mentioned in the literature as breakable dependencies [40].

[3]http://classycle.sourceforge.net/, http://depfind.sourceforge.net/, https://www.jarchitect.com/, https://github.com/clarkware/jdepend, http://lattix.com/, https://www.ndepend.com/, https://www.castsoftware.com/

– *Dense Structure* [32]: this smell arises when components have excessive and dense dependencies without any particular structure.
- **Violated principles:**
  – *Modularity*: the classed affected by this AS are usually overloaded with responsibilities. In fact they use a high number of other classes (incoming dependencies), i.e. they require a significant amount of information and/or functionalities that are not placed within the class. Furthermore they are used by a high number of other classes (outcoming dependencies), hence the information and/or the functionalities of the classes are required and most likely used by other classes. This scenario highlights a severe coupling problem between the hub-like class and all the others classes involved in the AS.
  – *Healthy Dependency Structure*: the severe coupling described above implies a dependency structure particularly unhealthy. That is because a change in a "hub" class will require to adapt at least all the dependent classes. Furthermore, the classes on which the "hub" class depends can be affected by the changes in the "hub" class and their change will affect the "hub" class. Therefore even the slight adjustment in a system affected by this smell will generate a ripple effect in all the involved classes.
- **Tools:** AI Reviewer, ARCADE, Arcan and Designite.
- **Detection Comparison:** the detection strategies are very different from each other due to the lack of an agreement regarding the number and the kind of dependencies required in order to detect this problem. In fact, AI Reviewer focuses the detection on concrete classes and the detection rule for this smell looks for structured types referencing more than 7 concrete (non-abstract) classes. While Arcan [13] focus the detection on abstraction and specifies that the dependencies have to be balanced, i.e. the difference between ingoing and outgoing dependencies must be less than a quarter of the total number of dependencies of the class. Instead, Designite[4] requires fun-in and fan-out metric values at least equal to 20. Finally ARCADE detects this smell if the dependencies number (ingoing and outgoing) is higher then the mean of the dependencies number (over all the system), plus the standard deviation of the dependencies number (over all the system), multiplied for a constant called `standard deviation factor`, which weights the contribution of the standard deviation (by default it is set to 1.5). Hence, according to the different metrics and threshold values adopted, the detection results can be largely different.

3. Unstable Dependency (UD)

- **Also known as:** Unstable Interface [23], which we describe as a variant.
- **Description:** Unstable Dependency describes a subsystem (component) that depends on other subsystems that are less stable than itself and because of this dependency the more stable files (classes) are changed frequently with the other files.
- **Variant:**
  – *Unstable Interface* [23]: this smell is focused on the stability of a set of specific files, which are the so-called Design Rule Spaces [23], i.e. the highly influential files of the system.
- **Violated principle:**
  – *Healthy Dependency Structure*: the dependencies that allow this smell to arise are extremely unhealthy because they increment significantly the change-proneness of the more stable subsystem and they also directly violate the Stable Dependencies Principle [38].
- **Tools:** Arcan, Designite and Hotspot Detector.
- **Detection Comparison:** all the smells that fall in this group implement the most common definition of this smell. However, some differences that concern the detection can be highlighted. In fact Hotspot Detector [23] evaluates the stability based on the frequency with which two files change together, according to the revision history. Instead Arcan [13] and Designite[4] relies on the Martin's Instability metric [38]. As consequence the detection results obtained through Hotspot Detector and the ones obtained using Arcan or Designite are likely to be different from each other.

4. Cyclic Hierarchy (CH)

- **Also known as:** Unhealthy Inheritance Hierarchy[5][23], Subtype Knowledge [33].
- **Description:** This AS refers to the mistake of directly referencing a subtype from a supertype. Therefore, it implies that there are circular dependencies between the namespaces containing sub- and supertype.
- **Violated principles:**
  – *Hierarchy*: inheritance should model a generalisation/specialisation hierarchy and the hierarchy specialisations should have dependencies that are directed only towards generalisations [15].
  – *Healthy Dependency Structure*: instability in the subtype will cause instability in the super-type, furthermore the supertype cannot be used and understood without its sub-type.
- **Tools:** AI Reviewer, Designite, Hotspot Detector and Massey Architecture Explorer.
- **Detection Comparison:** considered the specificity of the AS it is not strange to notice a high similarity in the definitions used by each tool. From the detection point of view, it is possible to see this AS as a very specific

[4] http://www.designite-tools.com/designite/does-your-architecture-smell/

[5] Hotspot Detector considers an inheritance hierarchy to be problematic when it falls into at least one of two cases [23]. In this catalogue, both these problems have been considered, however as two different ASs (Sections IV.4 and IV.7).

type of Cyclic Dependency (Section IV.1). Moreover, every tool states clearly that each occurrence of this AS should be refactored because it is far more likely to be an error or an oversight compared to any straightforward Cyclic Dependency, and this is why the AS is considered separately.

5. Scattered Functionality (SF)

- **Also known as:** Scattered parasitic functionality [20].
- **Description:** This smell arises when multiple components are responsible for realising the same high-level concern.
- **Violated principle:**
  - *Modularity*: This smell violates the principle of separation of concerns because the responsibilities are scattered across multiple components. Therefore, even if in this situation any modification that involves the "high-level concern" mentioned in the `Description` will generate an unhealthy chain of changes, the main problem is not related to the dependency structure but it is related to how the concerns are organised.
- **Tools**: ARCADE and Designite.
- **Detection Comparison:** The detection of this smell depends on the definition of concern on which the tool relies. In fact, each tool has to define the concept of "concern" not only in theory, but through some actually computable metrics. Designite highlights the concern by considering the external components with which the system interfaces. Accordingly, Designite[4] detects this smell by computing the number of accesses to external components and the default threshold for this metric is one. While the detection performed by ARCADE is based on the concept of "concern" introduced by Garcia et al. [41]. In fact, they define a "concern" as a role, responsibility, concept, or purpose of a software system. In order to group entities that handle similar system concerns into a single cluster, ARCADE leverages information retrieval techniques and machine learning algorithms [30].

6. God Component (GC)

- **Also known as:** God Class [29], Concern overload [20].
- **Description:** this smell indicates that a component implements an excessive number of concerns and accumulates too much control.
- **Violated principle:**
  - *Modularity*: this smell directly violates the principle of separation of concerns. In fact usually a God Component implies the presence of several satellite classes, that do not implement any functionality. This situation highlights a severe coupling problem between the God Component and the satellite classes and a severe cohesion problem within the God Component.
- **Tools**: AI Reviewer, ARCADE and Designite.
- **Detection Comparison:** the detection rule implemented by AI Reviewer[6] for this smell looks for classes which modify either directly or through a setter method least 3 data members (ex. fields) belonging to unrelated classes. A class is considered unrelated to the target class if it doesn't belong to the same class hierarchy and it doesn't contain the target class. While Designite[4] detects this smell at class level when is excessively large in terms of LOC, more than 27.000, and at package level when the number of classes is higher than 30. Finally ARCADE relies on a custom metric called *Number of Concerns per Component* [20]. The detection is accomplished by calculating the proportion of the *Number of Concerns per Component* and by checking if this number exceeds the *Concern Overload Topic Threshold* (fixed to 0.10 by default).

7. Abstraction without Decoupling (AwD)

- **Also known as:** Unhealthy Inheritance Hierarchy [5] [23].
- **Description:** This AS describes a situation where a client class uses a service represented as an abstract type, but also a concrete implementation of this service, represented as a non-abstract subtype of the abstract type.
- **Violated principle:**
  - *Healthy Dependency Structure*: the occurrence of this AS makes it difficult to replace the service implementation and to dynamically reconfigure or upgrade the systems. The client code must be updated, hence the client couples service description and service implementation together.
- **Tools:** Hotspot Detector and Massey Architecture Explorer.
- **Detection Comparison:** the main difference between the two detection rules is that Hotspot Detector [23] requires that the class depends on *all* the subtypes of the hierarchy, while Massey Architecture Explorer [33] requires that the class depends on at least one of the subtypes of the hierarchy.

8. Multipath Hierarchy (MH)

- **Also known as:** Degenerated Inheritance [33].
- **Description:** this smell arises when there are multiple inheritance paths connecting subtypes with their supertypes or a concrete class with their abstractions (abstract classes or interfaces).
- **Violated principle:**
  - *Hierarchy*: a hierarchical organisation of types helps to better understand the relationship between the types. When this smell occurs, it clutters the hierarchy and it increases the effort required to separate the sub- and superclasses. Therefore, the affected hierarchical structure is hard to understand and maintain [21].
- **Tools:** Designite and Massey Architecture Explorer.
- **Detection Comparison:** the specificity of the AS causes a high consistency in the definitions. The detection is

[6]http://www.aireviewer.com/doc/analyses/

accomplished by representing each hierarchy as a graph and by checking that the resulting graph is a tree. It is important to highlight that every tool states clearly that each occurrence of this AS should be refactored because it is quite likely to be the result of an error or an oversight operated during the implementation process.

9. Ambiguous Interface (AI)

- **Also known as:** Underused Interface [29].
- **Description:** This AS refers to the mistake of over-engineering an abstraction (an interface or a pure abstract class) by adding methods intended to accommodate potential future requirements.
- **Violated principle:**
  - *Modularity*: an extremely general abstraction promotes the creation of classes characterised by a high number of responsibilities, due to the large number of methods that the abstraction requires to be implemented (or the abstract class to be extended). Furthermore, the cohesion within the class decreases because of the "general purpose" approach used to define the abstraction.
- **Tools:** AI Reviewer and Designite.
- **Detection Comparison:** Designite considers this smell as a problem when there is only one entry-point to the system defined through that interface. While AI Reviewer is less restrictive and refers to an extremization of the code smell Speculative Generality [2] that causes the creation of very few general interfaces.

10. Unutilized Abstraction (UA)

- **Also known as:** Policy Detail Dependency [29], Supertype Bypass [29].
- **Description:** This AS refers to the mistake of directly referencing a concrete class or struct instead of referencing one of its supertypes from an abstract class or struct.
- **Violated principles:**
  - *Hierarchy*: one of the advantages of modelling a generalisation/specialisation hierarchy is the possibility to work at different levels of abstraction, instead of considering only concrete classes. The occurrence of this AS indicates that this advantage is not exploited properly.
  - *Healthy Dependency Structure*: the occurrence of this AS makes the client code depend on concrete implementations, making very hard to replace these implementations.
- **Tools:** AI Reviewer and Designite.
- **Detection Comparison:** this AS is detected in two different ways. The first detection strategy consists in finding the superclasses that do not have incoming dependencies. Both Designite and AI Reviewer detect the smell through this approach. However, the latter reports also as problematic the classes that depend on the subclasses instead of depending on the superclasses, through the smell Supertype Bypass [29].

11. Implicit Cross-module Dependency (ICD)

- **Also known as:** Implicit Cross Package Dependency [42], Logical Coupling [20].
- **Description:** Implicit Cross-module Dependency is a history based AS defined to compute the degree of co-changes occurring among files belonging to different packages detected by analysing the change history. This smell aims to captures hidden dependencies among files belonging to different packages. Hidden dependencies are co-change relations that can be found only in the history of the project [23]. However, some tools focus the attention on classes and packages instead of files.
- **Violated principle:**
  - *Healthy Dependency Structure*: the highlighted correlation between two, or more, classes that seem to change together can be employed to discern a single responsibility that has been split among more than one class. Therefore, if a change in that specific responsibility is required, all the classes need to be updated. Hence an unhealthy chain of changes has to be followed to correctly implement the modification.
- **Tools**: ARCADE, Arcan and Hotspot Detector.
- **Detection Comparison:** in order to accomplish this detection Hotspot Detector [23] relies on a Design Structure Matrix representation, i.e. a matrix that has the classes names on both rows and columns. In this matrix, the cell `(k,j)` contains the value concerning how many times the file `k` and the file `j` have changed together, according to the revision history. While in Arcan [42] the detection is accomplished through a graph representation, where both the elements (classes or packages) and the commits (that are stored in the Git repository) are represented as node and the relationships between them as edge. Therefore the time is discretized through the Git commits. In this situation two classes `k` and `j` are changed together if there are two `hasModified` edges: *(i)* one from a certain `commit` node to `k` and *(ii)* the other from the same `commit` node to `j`. Instead ARCADE [30] considers the architectural changes at two different levels: system-level and component-level. At the system-level, architectural change refers to the addition, removal, and modification of components (classes and interfaces). The detection of these changes is based on a custom metric called *a2a* (architecture-to-architecture) [30], which is defined as the distance between two architectures. At the component-level, architectural change reflects the placement of a system's implementation-level entity inside the architectural components (i.e. clusters) [30]. The detection of this changes is based on another custom metric called *c2c* (cluster-to-cluster) [43], which is defined as the degree of overlap between the implementation-level entities contained within two clusters.

12. Architecture Violation (AV)

- **Also known as:** Specification-Implementation Violation [44], Architecture Diagram Violation [37].

- **Description:** Architecture Violation aims to capture whether the intended architecture is different from its actual implementation.
- **Violated principle:**
  - *Healthy Dependency Structure*: we assumed that the intended architecture was "healthy", that's because it represents the design on which the software system was based. Accordingly with this assumption, every dependency that does not respect the intended architecture has to be considered as not healthy.
- **Tools:** Arcan, Sonargraph and Structure101.
- **Detection Comparison:** the similar aspect of all the detection strategies is the fact that the user has to manually specify the intended architecture. Therefore the main difference between the tools is related to how the intended architecture has to be specified. In Arcan [44], classes and packages are grouped by components and the components are linked by constraints edges according to the architecture specification. The user has to specify the components and the intended relations between them. The detection consists in checking the definition of the intended architecture, represented with rules and constraints in the dependency graph. While in Sonargraph the user has to write a `.arc` file in which he describes the intended architecture through a custom DSL (Domain-Specific Language)[7]. Finally, in Structure101 the user has to specify the intended architecture through Architecture Diagrams[8], which allow to define dependency constraints between arbitrary groups of code, which may or may not follow the physical structure. This AS, which has been already studied in the literature [45][46], is more laborious to be detected from the practitioners point of view. However, it is extremely useful to find those problems that are domain-specific or even project-specific.

A. Others architectural smells detected by Designite

Finally, it is important to outline that Designite detects a higher number of ASs, compared to the other tools considered in this work. Specifically, some ASs that underline problem of Modularization and Hierarchy that have been more specifically described in previous works [15], [21].

We have not described them, since according to our knowledge they are detected only by Designite and not by other tools. However, we list them for completeness: Feature Concentration, Broken Modularization, Insufficient Modularization, Wide Hierarchy, Deep Hierarchy, Rebellious Hierarchy, Unfactored Hierarchy, Missing Hierarchy. We can observe that Feature Concentration and Broken Modularization could be seen as variants of God Component, but the definitions of these two smells are quite different and for this reason we have not included them in the Variant field of God Component.

[7]http://blog.hello2morrow.com/2016/08/how-to-organize-your-code/
[8]https://structure101.com/help/java/studio5/\#perspectives/architecture-diagrams.html

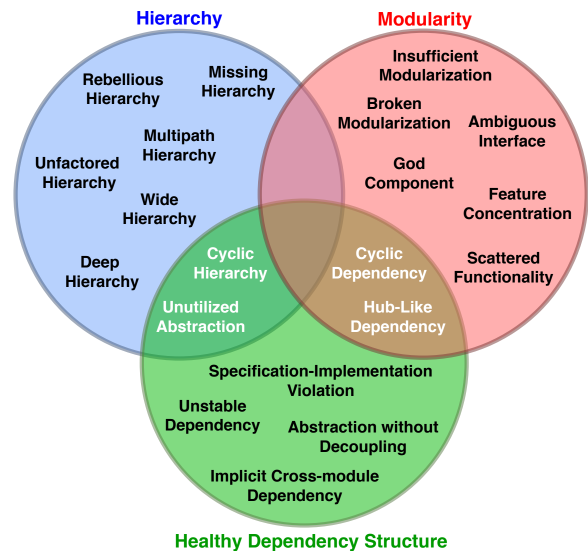

Fig. 2. Architectural smells classification

V. Architectural Smells Classification

In Section II we briefly described the catalogues proposed in the literature. Moreover, in Section III-C we described our proposal of classification based on the violation of three design principles.

With respect to the classification of Le et al. [20], the Interface and the Concern categories include ASs that violate the Modularity principle, which is the category chosen in our catalogue for those smells. Furthermore, it is possible to observe from the definitions of the categories proposed [20] that the ASs included in the Dependency category represent problems that affect the change-proneness of the classes and packages. Therefore, these ASs can be considered as a subset of the smells included in the Change category. In fact, in our catalogue, the Healthy Dependency Structure basically includes their two categories (Dependency and Change).

We also outline that some categories proposed by Lippert et al. [3] are similar to the ones proposed in this paper. Indeed, the smells in the Dependency category are closely related to the smells that violate our Healthy Dependency Structure principle, and the same happens for the Inheritance smells category and our Hierarchy category. The main difference between our classification and the one proposed by Lippert is represented by the category related to the Size. In fact, the smells that violate the Modulatity principle are not necessary smells related to the Size; an example is the Cycle Dependency AS which indicates a coupling problem but does not imply problems in terms of size of the packages or the classes involved. However, these two sets of smells are not completely disjointed, as it can be seen by taking into consideration the AS called God Component which violates the Modularity principle and implies a problem in terms of size of the affected class or package.

Finally, we have already extensively discussed the similarity and the differences between our classification and the one

proposed by Ganesh et al. [21] in Sections III-C and III-B.

In Figure 2 we show a diagram that represents our AS classification, also including the ASs mentioned in Section IV-A, which are the ones detected only by Designite. More work is still needed to better understand the impact of these smells on different quality issues and to extend the detection of the architectural problems/smells that actually negatively can affect the software quality of the projects.

According to our classification we outline that we classify only the AS that we introduced in our catalogue, hence only those for which there is a tool able to detect the smell. Obviously this classification proposal could be refined and extended. It is always difficult to identify the best classification, each proposal can provide useful hints according to specific aims. Obviously when new AS will be included in this catalogue, other improvements or new categories of the classification could be identified.

TABLE II
ARCHITECTURAL SMELLS DETECTED BY TOOLS

Tool name	1 CD	2 HLD	3 UD	4 CH	5 SF	6 GC	7 AwD	8 MH	9 AI	10 UA	11 ICD	12 AV
AI Reviewer	✓	✓		✓		✓			✓	✓		
ARCADE	✓	✓			✓	✓					✓	
Arcan	✓	✓	✓								✓	✓
Designite	✓	✓	✓	✓	✓	✓		✓	✓	✓		
Hotspot Detector	✓		✓	✓			✓				✓	
Massey Architecture Explorer	✓			✓			✓	✓				
Sonargraph	✓											✓
STAN	✓											
Structure 101	✓											✓

VI. CONCLUDING REMARKS

In this work, according to our previous experiences on AS detection and different tools experimentation ([25],[11],[12]), we inspected different tools in order to identify which AS the tools detect and then we deeply analysed them.

The ASs analysis has been accomplished by taking in consideration several aspects. The most interesting ones are the comparison of the detection strategies provided by the tools and the AS description, that has been obtained by the crossover of all the definitions taken in consideration by each tool. We observed that there is a lack of standardisation that concerns the ASs. In fact different names are often used to indicate the same problem, therefore the fields `Also known as` and `Variant` have been added to the AS template to deal with this inconsistency problem.

Furthermore, we propose a catalog of ASs (Section IV), where for each AS we reported its description, the `Violated principle` to describe the issues that the ASs imply, the tools that can be used to detect it and a description of the detection algorithms.

According to Q1, in Table II we outline the ASs detected for each tool that belong to the new catalogue that we proposed. It is interesting to note that the variance between the ASs detected is quite high, in fact most of them are detect only by four or less tools. Of course this consideration is not applicable to Cycle Dependency which is considered the most relevant and common AS [3][21]. Several studies shown how Cycle Dependency deeply affects software maintenance and it is the one considered more critical by developers [6]. In fact, all the tools detect this smell and the detection strategies are quite consistent with each others.

According to Q2, in Section 4 we outlined the differences in the detection rules/strategies adopted by each tool. We can observe that most of the AS that violate the Healthy Dependency Structure principle are detected through the dependency graph. Therefore, this data structure seems to be quite interesting and it is possible that some additional inference concerning the architectural quality assessment can be accomplished through it. While it is far more intuitive that the detection of the ASs that violate the Hierarchy principle is based on the hierarchy graphs. Furthermore, it is possible to notice how the detection strategies of the AS that violate the Modularity and the Healthy Dependency Structure principles are often related to the concept of "concern", i.e. a software system's role, responsibility or purpose [47]. We also underlined how the detection of the concerns is accomplished in different ways, and it is important to highlight that all the approaches could be improved or merged together in order to reach higher values of precision during the AS detection.

According to Q3, we proposed a classification in three categories based on the violation of three design principles, and we observed that some smells belong to more categories.

In Section V, we outlined the main differences and similarity between our classification and the ones presented in Section II.

The main issue that we have noticed with the classifications proposed by Ganesh et al. [21] and Lippert et al. [3] is that they include in the catalogue smells that are more close to the concept of code smell, which correspond to the smells that violate the Abstraction and Encapsulation principles for the former and the smells related to the Size for the latter. While the classes proposed by Le et al. [20] were characterised by fuzzy boundaries, in fact we underlined how the smells belonging to different categories actually share several aspects from the violated principles point of view. Hence, our classification has as distinguishing features the fact that is focused only on the classification of ASs and that is more closely related to the implementation of the AS detection. These two features allow a more distinct and consistent classification of the smells taken into account in the catalogue.

We aim to extend and refine the catalogue by considering other detector tools and AS currently not included since not available or not found. We aim also to refine the classification or to propose other kinds of classification. In particular, it would be interesting to identify classes of AS which impact on specific quality attributes, such as performance and security or other quality attributes. We aim to extend the AS template in the catalogue by considering also possible refactoring suggestions for each AS. Finally, we are considering to extend this catalog to microservices smells [48] and antipatterns [49] when the tool support to detect them will be avaialble.

REFERENCES

[1] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Joint Meeting on Foundations of Sw. Engineering, ESEC/FSE*, 2015.

[2] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Component software series, Addison-Wesley, 1999.

[3] M. Lippert and S. Roock, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, May 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 *European Conference on Software Maintenance and Reengineering*, pp. 277–286, March 2012.

[5] Y. Cai and R. Kazman, "Detecting and quantifying architectural debt: theory and practice," in *Intl. Conf. on Software Engineering, ICSE*, 2017.

[6] A. Martini, F. Arcelli Fontana, A. Biaggi, and R. Roveda, "Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company," in *Proc. of the European Conf. on Software Architecture (ECSA)*, (Madrid, Spain), Springer, Sep. 2018.

[7] R. Verdecchia, I. Malavolta, and P. Lago, "Architectural technical debt identification: The research landscape," in *International Conference on Technical Debt*, TechDebt '18, 2018.

[8] N. Ali, S. Baker, R. O'Crowley, S. Herold, and J. Buckley, "Architecture consistency: State of the practice, challenges and requirements," *Empirical Software Engineering*, vol. 23, pp. 224–258, Feb 2018.

[9] V. Lenarduzzi, A. Sillitti, and D. Taibi, "Analyzing forty years of software maintenance models," in *39th International Conference on Software Engineering Companion*, ICSE-C '17, IEEE Press, 2017.

[10] V. Lenarduzzi, A. Sillitti, and D. Taibi, "A survey on code analysis tools for software maintenance prediction," in *Software Engineering for Defence Applications SEDA*, 2019.

[11] F. Arcelli Fontana, R. Roveda, M. Zanoni, C. Raibulet, and R. Capilla, "An experience report on detecting and repairing software architecture erosion," in *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 21–30, April 2016.

[12] F. Arcelli Fontana, R. Roveda, S. Vittori, A. Metelli, S. Saldarini, and F. Mazzei, "On evaluating the impact of the refactoring of architectural problems on software quality," in *Proceedings of the Scientific Workshop Proc. of XP2016, Edinburgh, Scotland, UK, May 24, 2016*, p. 21, 2016.

[13] F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic detection of instability architectural smells," in *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, pp. 433–437, 2016.

[14] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[15] S. Ganesh, T. Sharma, and G. Suryanarayana, "Towards a principle-based classification of structural design smells," *Journal of Object Technology*, vol. 12, pp. 1:1–29, June 2013.

[16] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.

[17] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *Journal of Object Technology*, vol. 11, no. 2, pp. 5: 1–38, 2012.

[18] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11, pp. 395–431, Sep 2006.

[19] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*, QoSA '09, 2009.

[20] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic, "Relating architectural decay and sustainability of software systems," in *Working Conference on Software Architecture (WICSA)*, pp. 178–181, 2016.

[21] G. S. Girish Suryanarayana and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*. Burlington, Massachusetts, USA: Morgan Kaufmann, 2014.

[22] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., 1st ed., 1998.

[24] C. Baldwin and K. Clark, *Design Rules Volume I: The Power of Modularity*. MIT Press, 2000.

[23] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *Working Conf. on Software Architecture*, 2015.

[25] F. A. Fontana, I. Pigazzini, R. Roveda, D. A. Tamburri, M. Zanoni, and E. D. Nitto, "Arcan: A tool for architectural smells detection," in *2017 IEEE International Conference on Software Architecture Workshops, Gothenburg, Sweden, April 5-7, 2017*, pp. 282–285, 2017.

[26] CAST©, "CAST." https://www.castsoftware.com/, accessed 2019-01.

[27] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, "Supporting the identification of architecturally-relevant code anomalies," in *International Conference on Software Maintenance (ICSM)*, 2012.

[28] L. Xiao, Y. Cai, and R. Kazman, "Titan: A toolset that connects software architecture with quality analysis," in *Intern. Symposium on Foundations of Software Engineering*, FSE 2014, pp. 763–766, 2014.

[29] Logarix, "AI Reviewer." www.aireviewer.com, accessed 2019-01.

[30] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural change in open-source software systems," in *Working Conference on Mining Software Repositories*, pp. 235–245, May 2015.

[31] ESSeRE Lab, Università degli Studi di Milano Bicocca, "Arcan." http://essere.disco.unimib.it/wiki/arcan, accessed 2019-01.

[32] T. Sharma, P. Mishra and R. Tiwari, "Designite." www.designite-tools.com, accessed 2019-01.

[33] Jens Dietrich, Massey University, "Massey Architecture Explorer." http://xplrarc.massey.ac.nz/, accessed 2019-01.

[34] J. Dietrich, "Upload your program, share your model," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, 2012.

[35] hello2morrow, "Sonargraph." www.hello2morrow.com, accessed 2019.

[36] Bugan IT Consulting UG, "Structural Analysis for Java." http://stan4j.com/, accessed 2019-01.

[37] Headway Software Technologies Ltd., "Structure101." http://structure101.com/, accessed 2019-01.

[38] R. C. Martin, "Design principles and design patterns," *Object Mentor*, vol. 1, no. 34, p. 597, 2000.

[39] J. Dietrich, C. McCartin, E. Tempero, and S. M. A. Shah, "On the existence of high-impact refactoring opportunities in programs," in *Australasian Computer Science Conference*, ACSC '12, 2012.

[40] F. Arcelli Fontana and S. Maggioni, "Metrics and antipatterns for software quality evaluation," in *Proceedings of the 34th IEEE Software Engineering Workshop (SEW 2011)*, (Limerick, Ireland), pp. 48–56, IEEE, June 2011.

[41] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *International Conference on Automated Software Engineering (ASE 2011)*, Nov 2011.

[42] R. Roveda, F. Arcelli Fontana, I. Pigazzini, and M. Zanoni, "Towards an architectural debt index," in *Proceeding Euromicro Conference, SEAA-TD, session on Technical Debt, Prague*, SEaTeD '18, 2018.

[43] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *International Conference on Automated Software Engineering*, ASE'13, 2013.

[44] A. Biaggi, F. Arcelli Fontana, and R. Roveda, "An architectural smells detection tool for c and c++ projects," in *Proceeding Euromicro Conference, SEAA-TD, session on Technical Debt, Prague*, SEaTeD '18, 2018.

[45] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: bridging the gap between design and implementation," *IEEE Transactions on Software Engineering*, vol. 27, pp. 364–380, April 2001.

[46] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-oriented software architecture, on patterns and pattern languages*, vol. 5. John wiley & sons, 2007.

[47] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *International Conference on Automated Software Engineering (ASE 2011)*, 2011.

[48] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018.

[49] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: a systematic mapping study," *8th International Conference on Cloud Computing and Services Science (CLOSER2018)*, 2018.