Towards assessing software architecture quality by exploiting code smell relations

Francesca Arcelli Fontana*, Vincenzo Ferme†, Marco Zanoni*
*Department of Informatics, Systems and Communication
University of Milano-Bicocca, Milano, Italy
Email: arcelli@disco.unimib.it, marco.zanoni@disco.unimib.it
†Faculty of Informatics
University of Lugano (USI), Switzerland
Email: vincenzo.ferme@usi.ch

***Abstract*—We can evaluate software architecture quality using a plethora of metrics proposed in the literature, but interpreting and exploiting in the right way these metrics is not always a simple task. This is true for both fixing the right metric threshold values and determining the actions to be taken to improve the quality of the system. Instead of metrics, we can detect code or architectural anomalies that give us useful hints on the possible architecture degradation. In this paper, we focus our attention on the detection of code smells and in particular on their relations and co-occurrences, with the aim to evaluate tecnical debt in an architectural context. We start from the assumption that certain patterns of code anomalies tend to be better indicators of architectural degradation than simple metrics evaluation.**

I. Introduction

For software quality assessment, we can use different kinds of metrics at code level, as the well known CK suite metrics form Chidamber and Kemerer [1] and metrics at design level, as those proposed by Martin [2]. Many tools have been developed for the computation of these metrics, some of them provide also different kinds of useful views of the system under analysis. In any case, as already asserted [3], there is no standard set of base measures to assess software architecture quality.

Moreover, when we get the value of metrics, we have to interpret them, face the problem of thresholds setting and identify when a value has to be considered critical or not. Different authors addressed this problem in the literature (i.e. [4], [5]), but a widely accepted proposal has not been reached yet. Moreover, also if we know that the values of some metrics are critical according to the fixed thresholds, which actions can we immediately take to improve the quality of the system? Which easily implementable suggestions can we get?

In this perspective, instead of looking at metrics, we can look at code, design or architecture problems, which can be identified through metrics or in other ways, but for which we can get more direct and exploitable information on how to solve them. Examples of these anomalies are code smells, structural antipattern, and architecture smells.

Code smells are symptoms of possible code or design problems, that can be removed through refactoring. Fowler defined 23 code smells [6] and other smells have been identified in the literature later, classified according to different taxonomies. Among them, it is important to focus on those that could be more critical for code and architecture quality. Example of code smells are given in Section III.

Structural antipatterns are software structures and entities that seem to be an adequate solution to a certain design or programming issue, but they are actually far from the optimal implementation. The presence of antipatterns inside an object-oriented system reflects a system being not modular, far from the object-oriented best practices, and difficult to maintain and reuse. Structural antipatterns help to identify components of a system that are critical in terms of their structure, i.e., in terms of their number of outgoing and incoming relationships with the rest of the system. Examples of antipatterns include butterfly, hub, breakable and tangle antipatterns (see for their description [7], [8]).

Architecture smells are commonly used architectural decisions that negatively impact system quality and are caused by a violation of recognized design principles. Architecture smells point to places in the system's architecture that should be further analyzed. Both the two terms architecture [9] and architectural [10] smells are used. In the following, we call them architectural smells. Lippert [9] identified architectural smells at various levels, in and between packages, subsystems and layers, and smellls in inheritance hierarchies. While Garcia et al. [10] define the Connector Envy, the Scattered Fuctionaity, the Ambiguous Interface and the Extraneus Connector architectural smells.

As outlined by Macia et al. [11], some code anomalies, as code smells, and recurring patterns of them could be key indicators of architecture degradation, and they call them *architecturally relevant* code anomalies.

In this paper, we focus our attention on the possible relations existing among code smells and their co-occurrence (how many entities are affected by more than one smell), with the aim to find and detect some architecturally relevant code anomalies. The information we can capture in this way is useful to focus our attention on the most critical smells and to find a way to prioritize them for their removal through the right refactoring steps. For example, if a smell is related to many other smells, this can be considered more critical than an "isolated smell".

978-1-4673-7076-9/15 $31.00 © 2015 IEEE
DOI 10.1109/SAM.2015.8

An other interesting aspect related to code smells for improving architecture quality, and also outlined by Vidal et al. [12], is that if the developer is interested in improving, for example, the coupling of the system, he can focus on the analysis and then removal of particular smells as Intensive Coupling, Dispersed Coupling and Shotgun Surgery smells [6], because these smells impact on coupling aspects. While if he wants to improve cohesion, he has to consider first God Class, Brain Method or Intensive Coupling smells. Hence, indirectly we can get through the detection of smells more useful information on how to improve the system, respect to take into account only metric values.

In the paper, we also propose three Views and a specific *Related Code Smell View*, useful to deeply investigate the potential problems pointed out by the presence of the detected code smells. The set of personalized views for each code smell show details about the code smell and its impact on system design quality.

Hence, our ongoing and future investigation, is towards the study of the possible relations existing among code smells and architectural smells or antipatterns. Probably, if we locate an architectural smell, we can identify some related code smells, or better, vice versa, the presence of some code smells could imply the presence of a particular architectural smell or antipattern. We are interested in this last direction of investigation, because directly identifying architectural smells is difficult for the lack of automatic tool support.

The principal aims of the paper regard:

- the possibility to identify patterns of related or co-occurred smells integrated in the code smell detection process that we have developed (Section III);
- our experimental evaluation of related and co-occurred smells in 74 systems of the Qualitas Corpus of Tempero et al. [13] (Section IV);
- the definition of a Related Code Smell View useful to immediately get an impression on the criticality of the smell, its relation with other smells and on the identification of its refactoring opportunities (Section V);
- the steps of our ongoing research towards the identification of architectural smells through code smells relations (Section VI).

II. Related work

For what concerns architectural smells, we have cited in the Introduction two well known works of Garcia et al. and Lippert et al.

Vidal et al. [12] propose an approach to prioritize critical code smells for refactoring, by taking into account aspects that can compromise the architecture of a system, as related modifiability scenarios [14], past component modification and the relevance of the smell for the developer. Macia et al [11] and Arcoverde et al. [15] start addressing the problem that a code smell is important if it compromises the architecture of the system. By removing them, the degradation of the architecture could be stopped. Their analysis has been done through the involvement of expert developers for the identification of the architectural problems. As outlined by the same authors, state of art mechanisms are not accurate to detect architecturally-relevant code anomalies. Moreover, they assert that "certain patterns of code anomalies tend to be better indicators of architectural degradation symptoms than single code anomalies. However, these patterns cannot be directly detected by strategies, which focus on identifying individual code anomalies." The relevance of the existence of more anomalies has been outlined also by Abbes et al. [16] for Blob and Spaghetti code antipatterns.

In our paper, we outline that we are able to detect not only the individual smells, but the related code smells, integrated in our detection approach. We also provide some Views that can be useful to better explore the relations and the impact of smells on software architecture quality.

III. Code Smell Relations

We defined six code smells detection rules based on the computation of metrics. We computed a total of 43 metrics, to be used for code smell detection and other software quality assessment tasks. These metrics are well-known software quality metrics defined in the literature, with the addition of some ad-hoc metrics, defined for the detection of particular smells. Metric thresholds used in the detection strategies were derived through an automatic method we have defined ([17]).

We are currently able to detect six smells [6], [18]: God Class, Data Class, Brain Method, Shotgun Surgery, Dispersed Coupling and Message Chains. We decided to focus our attention on these smells because they are among the most related to faults or change-proneness and the most common [19].

A. Code Smells Definitions

We provide the definitions of the analyzed smells according to those proposed in the book [18], plus the definition for the Message Chains smell [6].

God Class refers to classes that tend to centralize the intelligence of the system. A God Class performs too much work on its own, delegating only minor details to a set of trivial classes and using the data from other classes.

Data Class refers to classes that store data without providing complex functionality, and having other classes strongly relying on them. A Data Class reveals many attributes, it is not complex, and exposes data through accessor methods.

Brain Method tends to centralize the functionality of a class, in the same way as a God Class centralizes the functionality of an entire subsystem, or even a whole system.

Shotgun Surgery refers to a method that implements a functionality used by a large number of other classes and methods. Changes to this method imply many changes to many other methods in a large number of classes.

Dispersed Coupling appears when an operation is excessively tied to many other operations in the system, and then these provider methods are dispersed among many classes.

Message Chains occurs when a client asks one object for another object, which the client then asks for yet another object, and so on.

Table I
METRICS USED FOR CODE SMELLS DETECTION

Short Name	Long Name
LOCNAMM	Lines of Code Without Accessor or Mutator Methods
LOC	Lines Of Code
WMCNAMM	Weighted Methods Count of Not Accessor or Mutator Methods
NOMNAMM	Number of Not Accessor or Mutator Methods
TCC	Tight Class Cohesion
ATFD	Access To Foreign Data
WOC	Weight Of Class
NOAM	Number Of Accessor Methods
NOPA	Number Of Public Attributes
CYCLO	McCabe Cyclomatic Complexity
MAXNESTING	Maximum Nesting Level
NOLV	Number Of Local Variables
ATLD	Access To Local Data
CC	Changing Classes
CM	Changing Methods
FANOUT	Number of Called Classes
CINT	Coupling Intensity
CDISP	Couplind Dispersion
MaMCL	Maximum Message Chain Length
MeMCL	Mean Message Chain Length
NMCS	Number of Message Chain Statements

Not necessarily all the identified code smells in a system represent problems, but simply give hints on them. For example Java Beans are Data Classes, but they are not really a problem. For this reason, we defined in [20] some kind of filter to discard false positive instances of smells.

We provide below an example of the detection rule for the God Class, where all the metrics used in the detection of the six smells are reported in Table I and the values for the thresholds (LOW, MEAN, HIGH) are derived through an automatic method we have defined [17].

A God Class performs too much work on its own (LOCNAMM $\geq$ HIGH, WMCNAMM $\geq$ MEAN, TCC $\leq$ LOW), delegating only minor details to a set of trivial classes (NOMNAMM $\geq$ HIGH) and using the data from other classes (ATFD $\geq$ MEAN). Here we decided to use the LOCNAMM (Lines of Code Without Accessor or Mutator Methods) metric instead of the LOC one, because getter and setter methods are often generated by the IDE. A class having getter and setter methods and a class without getter and setter methods must have the same chance to be detected as God Class. We did the same choice for the WMCNAMM (instead of WMC) and NOMNAMM (instead of NOM) metrics. Hence, the detection rule is:

(LOCNAMM $\geq$ HIGH) $\wedge$ (WMCNAMM $\geq$ MEAN) $\wedge$ (NOMNAMM $\geq$ HIGH) $\wedge$ (TCC $\leq$ LOW) $\wedge$ (ATFD $\geq$ MEAN).

A detailed description of the detection rules of the six smells can be found in [19].

B. Related Code Smells

A code smell can be related to other code smells. This aspect has been first investigated by Pietrzak et al. [21] and other authors addressed this issue later (e.g. [22], [23]).

Table II
STATISTICS ABOUT THE 74 QUALITAS CORPUS SYSTEMS

LOC	N. Packages	N. Classes	N. Methods
6,785,568	3,420	51,826	404,316

As part of our detection approach, we also considered code smell relations, and we tried to identify if some relations among smells exist and occurr frequently. For the six smells that we are able to detect, we defined different sets of related elements:

- *Contained smells*: for every code smell, the set of all other smells contained in the same class/method, e.g., a Brain Method contained in a God Class;
- *Calling smells*: for God Class, Data Class, Brain Method, Shotgun Surgery, the set of other smelly classes/methods calling the considered smell;
- *Called smells*: for Dispersed Coupling, the set of other smelly classes/methods that are called by the smelly method;
- *Used smells*: for God Class, the set of Data Classes it uses (accesses data).

We decided to consider and detect these relations among smells because they are useful to understand the smells, their criticality and to find the best way to refactor them [6], [24]. Other relations among smells have been proposed in the literature (e.g., [18], [21]), that we can consider for future developments, also by extending the number of smells detected by our tool.

Moreover, we consider and detect the co-occurrence of smells, as a specialization of the first relation described above. We define the co-occurrence of code smells in this way: two or more smells co-occur if they are detected together in the same class or the same method. In particular, we will consider and report in Section IV cases of two and three co-occurrent smells.

IV. EXPERIMENTAL EVALUATION

We identify related code smells and co-occurrences of code smells, reporting how they are distributed on the reference dataset, which contains a large number of systems, heterogeneous in terms of domains, functionalities and size. Table II summarizes some statistics about the dataset of the analyzed systems of the Qualitas Corpus, a curated collection of 111 open source systems [13] classified in different categories. We selected 74 systems that were possible to compile, and we added missing libraries when required.

In Table III, we outline the number of smells detected in the 74 systems at class and method level. We checked also how the affected classes span over the 74 systems and we observed that Data Class is present in 96% of the systems, and Brain Method and Dispersed Coupling affect more than 90% of the systems, while Shotgun Surgery and Message Chain more than 50%.

Table III
CODE SMELLS DETECTED ON 74 SYSTEMS

Code Smell	Affected		Systems #	Systems %
God Class	808 classes	1.56%	61	82
Data Class	1,605 classes	3.10%	71	96
Brain Method	2,701 methods	0.67%	69	93
Shotgun Surgery	497 methods	0.12%	49	66
Dispersed Coupling	3,188 methods	0.79%	68	92
Message Chains	634 methods	0.16%	39	53

Table IV
RELATED CODE SMELLS (CSS) STATISTICS ON 74 SYSTEMS

Relation	Max	> 0 (#)	> 0 (%)
God Class			
Data Classes used by the God Class	12	207	25.62
Classes with CSs that call the God Class	41	468	57.92
Other CSs inside the God Class	25	464	57.43
Data Class			
Classes with CSs that call the Data Class	9	307	38.00
Brain Method			
Other CSs inside the Brain Method	2	266	9.85
CSs on the class that contains the Brain Method	1	432	15.99
Classes with CSs that call the Brain Method	5	137	5.07
Shotgun Surgery			
Classes with CSs that call the Shotgun Surgery	11	262	52.72
CSs on the class that contains the Shotgun Surgery	1	150	30.18
Other CSs inside the Shotgun Surgery	2	53	10.66
Dispersed Coupling (DCO)			
Classes with CSs called by the DCO	47	2,242	70.33
CSs on the class that contains the DCO	1	531	16.66
Other CSs inside the DCO	2	254	7.97
Message Chains			
Other CSs inside the Message Chains	2	172	27.13
CSs on the class that contains the Message Chains	1	131	20.66

A. Related Code Smells

Table IV reports the results of the analysis we performed to assess the code smell relations (defined in Section III-B) existing among the supported code smells. For each relation, we consider some descriptive statistics:

- *Max*: the maximum number of entities found in the relation of a single code smell instance;
- > 0 *(#/%)*: the number/percentage of entities having a relation.

As an example, from the first line of Table IV we can read that 207 God Classes (that are 25.62% of the total God Classes we found) access data of at least one Data Class. Moreover, the maximum number of Data Classes used by a single God Class is 12. In the following, we report some considerations regarding each of the considered smells.

1) God Class: 26% of God Classes access data from Data Classes. A large number (58%) of entities affected by God Class are called by at least one class affected by code smells. 57% of the God Classes have at least one method affected by at least one of the method code smells we investigated. God Classes attract many smells, and single God Classes can be related to many other smells, as we can see from the Max values. When dealing with a God Class, it is high probable that other smells will be present in the same class or in the coupled ones.

2) Data Class: Among Data Classes, 38% are used by other classes affected by code smells. We investigated these classes and found that God Class affects most of them. The link between God Class and Data Class is known in the literature, and confirmed by our results.

3) Brain Method: The number of Brain Methods affected also by other code smells is low (9.85%); approximately 16% of Brain Methods are in classes affected by code smells (mainly God Class), and are called seldom by entities affected by code smells.

4) Shotgun Surgery: It is called by a large number of classes affected by code smells. This smell identifies those methods that are called by a large number of classes; our analysis shows that 53% of the classes calling a Shotgun Surgery are affected by at least one code smell. 30% of classes that contain at least a method affected by Shotgun Surgery are affected by a code smell. Only the 11% of the Shotgun Surgery methods are affected also by other code smells.

5) Dispersed Coupling: Very often (70% of the times), classes called by Dispersed Coupling methods are affected by code smells, and a single Dispersed Coupling can call many other smelly classes. Approximately 17% of the classes that contain Dispersed Coupling are affected by other code smells, while only the 8% of the methods affected by Dispersed Coupling are affected also by other code smells.

6) Message Chains: Approximately 27% of Message Chains methods are affected by at least another code smell, and 21% of the classes that contains Message Chains are affected by code smells.

The defined relations can be exploited in a significant number of cases, as witnessed by the reported data. The user can exploit these relations to enhance the refactoring process of a given smell. Table IV shows that a code smell is often not isolated. It is common having at least another related code smell, and this happens in nearly all the systems we investigated. It is important to capture and to show these relations to the users, because they are fundamental to better understand the impact of code smells in the systems, and decide the best way to remove them. Consider for example a God Class: knowing if it uses some Data Classes can be relevant to understand how to refactor the God Class. We can for example move some methods from the God Class to the Data Classes it uses, to remove both God Class and Data Classes.

B. Code Smells Co-Occurrence

Table V reports methods code smells co-occurrence (more than one smell affecting the same entity), that summarizes a subset of the information regarding the related code smells discussed above, and in particular the ones in the form of

Table V
METHODS CODE SMELLS CO-OCCURRENCE ON 74 SYSTEMS

	entities N.	%	Brain Method	Shotgun Surgery	Dispersed Coupling	Message Chain
2 smells	161	2.73%	×	·	×	·
	85	2.55%	×	·	·	×
	78	2.04%	·	·	×	×
	14	1.24%	·	×	·	×
	23	0.72%	×	×	·	·
	20	0.54%	·	×	×	·
3 smells	3	0.07%	·	×	×	×
	2	0.03%	×	·	×	×
	1	0.02%	×	×	×	·

"Other code smells inside the given code smell". The co-occurrence of the two code smells affecting classes (God Class and Data Class) is not reported because it is zero; it is an expected result, as they describe classes with completely different characteristics. The table lists first the co-occurrence of two code smells, and then the co-occurrence of three code smells, sorted by descending number of found instances.

In the reference dataset, at least two code smells affect approximately 1–2% of the smelly methods. The most common co-occurrence couples are Brain Method ↔ Dispersed Coupling and Brain Method ↔ Message Chains. Brain Methods tend to centralize the intelligence of the classes that contain them. Often, this means that they tend to interact with many classes, resulting in the co-occurrence with two code smells impacting on coupling. The co-occurrence of three code smells is very low, and is zero if we consider all the four code smells.

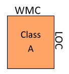
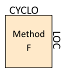
V. CODE SMELLS RELATIONS VIEWS

We have developed a visualization framework driven by the following considerations:

1) each code smell has specific characteristics, quantified by the metrics used in the detection strategy;
2) a code smell is sometimes related to other code smells, as outlined in the previous section.

We decide to use the polymetric view paradigm proposed by Lanza [25], where vertices are represented by rectangles with a certain width, height and colour. The width and height are proportional to some software metrics related to the entity represented by the vertex. The vertex can be used to show "smelly" and not "smelly" entities. For the not "smelly" entities, we use the white colour. For "smelly" entities, we use a graduated scale of colours that span from light orange to dark orange. Each colour of the scale corresponds to a specific intensity level of the smell. We have identified an Intensity level Index (not described in this paper) to prioritize the most critical code smells; the index is based on the metrics used in the detection rule of the smell, whose values exceed the fixed thresholds, and summarizes metrics values in a single score in the range 1–10. We use vertices to represent classes and methods. Table VI shows an example of metrics associated to the vertices.

Table VI
EXAMPLE OF METRICS FOR CLASSES AND METHODS VERTICES

Vertex	Name	Smell Label	Height Metric	Width Metric	Colour
	Class A	God Class (GC)	LOC	WMC	Problem
	Method F	Brain Method (BM)	LOC	CYCLO	Warning

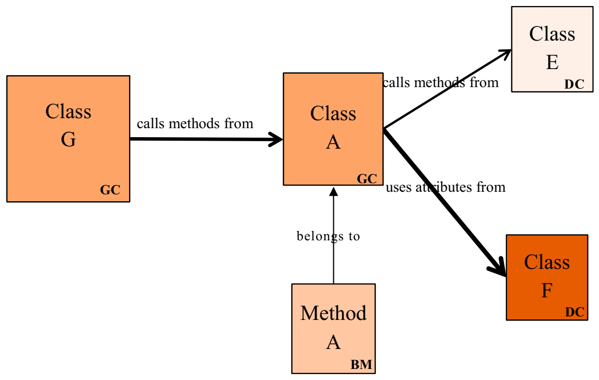


Figure 1. Related God Class view example

We use edges to represent Relations between classes and methods, represented by an arrow with a weight and a label. The label describes the kind of the represented relation, while the weight is associated to the strength of the relation (e.g., the number of attributes used by a class from another one). Graphically, a more weighted relation makes the arrow thicker (e.g., the greater the number of attributes used by a class from another one, the thicker the arrow is). Both the weight and the label are optional. We define three types of Relations: belonging, hierarchy and coupling relation. The belonging relation connects an entity to its container (e.g., a method belonging to a class). It has the label "belongs to" and no weight. The hierarchy relation binds a class to its superclass. It has no label and no weight. The coupling relation connects two entities that are coupled. It has a label and a weight.

We propose three views that show the code smell characteristics and their impact on the system design. We characterize these three views for each of the six code smells we detected.

1) *Code Smell Profile*: shows simply the characteristics of the source code entity captured by the code smell.
2) *Code Smell Context View*: enriches the previous one with some entities and relations, to better locate the code smell in the context of the system it belongs to; the view is useful to find out the interaction of the code smell with other parts of the system.
3) *Related Code Smell View*: offers a visualization of the related code smells.

Figure 1 shows an example of the *Related Code Smell View*, where the God Class (Class A, with label: GC) is on the centre of the screen with the different related code smells, i.e., Data Class (label: DC) and Brain Method (label: BM). We can observe in the figure that the edges have different weights associated to the strength of the relation; for example the strength of the relation between Class G and Class A is proportional to the number of called methods from G to A. The figure displays different kinds of relations:

- Data Classes used by the God Class: the God Class is connected to Data Classes through generic relations of three possible kinds: "uses attributes from", "calls methods from" or "uses attributes and calls methods from"; in the figure, Class A calls methods of the Class E Data Class and uses attributes from the Class F Data Class.
- "smelly" classes calling the God Class: they are connected to the God Class through generic relations of three possible kinds: "uses attributes from", "calls methods from" or "uses attributes and calls methods from"; in the figure, Class G, affected by God Class, calls methods from Class A.
- Other code smells inside the God Class (e.g., Brain Method): if the same class is affected by other code smells, the respective label is added to the GC label; for "smelly" entities contained in the God Class (e.g., methods), a belonging relation connects the entities with the God Class; in the figure, Method A, affected by Brain Method, belongs to Class A.

We have not implemented yet these views in our tool, but we already stored in our model all the information needed to realize them.

VI. Discussion on Ongoing Research

We are able to detect six code smells and some relations existing among them. We would like to exploit this information towards the detection of possible architectural anomalies.

Not all code anomalies are dangerous at architecture level, and we have to find a way to locate the more critical ones. For this reason, we aim to study if the detection of correlated smells, as described in this paper, could be used to identify architectural anomalies, also by exploiting the Related Code Smell View described above. To this aim, we will extend the set of detected code smells, and detect the corresponding code smell relations, co-occurrences and views. We will start by focus our attention on some few Architectural smells as for example those studied by Garcia et al. [26]

As outlined by Macia et al. [11], certain recurring patterns of co-occurring code anomalies tend to be stronger indicators of architectural degradation symptoms. For instance, co-occurrences of Long Method and Divergent Change smells were associated with architectural problems in all the systems they studied. The Long Method smell can be seen as a simplified version of the Brain Method smell, while the Divergent Chain smell [6] occurs when one class is commonly changed in different ways for different reasons. Hence, we can find different chains of inter-related smells that could be exploited to identify architectural anomalies.

Table VII
Code smells impact on Metrics categories

Code Smell	Affected entity	Intra/Inter	Impacted OO Quality Dimensions
God/Large Class	Class	Intra Class	Coupling, Cohesion, Complexity, Size
Brain/Long Method	Method	Intra Class	Coupling, Cohesion, Complexity, Size
Shotgun Surgery	Method	Inter Class	Coupling
Message Chains	Method	Inter Class	Coupling
Dispersed Coupling	Method	Inter Class	Coupling, Cohesion, Complexity
Data Class	Class	Intra Class	Encapsulation, Data Abstraction

We aim to investigate this aspect, through the following steps:

- detect new code smells, that could be more useful according to software architecture evaluation, as for example the following smells of Fowler: Divergent Change, Parallel Inheritance Hierarchies, Alternative Classes with Different Interfaces and Incomplete Library Class smells;
- identify new code smell relations and detect the related and co-occurred smells;
- check if these relations and the corresponding views could be used to outline the presence of some specific architectural smells, as those described by Garcia;
- take into account the evolution of software projects, both as a technique for detecting existing code smells, e.g., Divergent Change, or to define new smells capturing the erosion of the architecture of the entire project or its individual modules;
- consider also structural antipatterns as those outlined in the Introduction, where their detection is essentially based on dependency metrics computation. We have already developed the automatic tool support to detect these antipatterns [7].

Moreover, as we outlined in the Introduction, according to the interest a maintainer/developer could have in improving a particular category of metric (e.g., cohesion), he can take into account some smells prior than to others. In Table VII, we outline the impact that our six considered smells have on some metric categories. If one is interested to check if the software architecture satisfy the best practice of modularization with high cohesion and low coupling, he can start considering first the code smells which have impact on cohesion and coupling respect to other ones.

VII. Conclusions

We defined and identified different kinds of relations among code smells, and measured their frequency in a reference dataset, composed of 74 systems. We found that a significant percentage of the detected instances has a relation with other instances. For example, we found that 26% of God Classes use at least a Data Class, and that 53% and 70% of methods affected by Shotgun Surgery and Dispersed Coupling, respectively, are called from (at least) one class or method affected by a code smell. This observation confirms other results and theories proposed in different studies, suggesting that code smells tend to cluster together and interact in many ways, and that clusters of smells have more effect on maintainability than isolated smells.

We investigated in more depth one basic form of code smell relation, i.e., the co-occurrence of code smells. We measured how many times single classes (or methods) are affected by more than one smell at the same time. We found no co-occurrence between God Class and Data Class. The result was expected since the two smells are extremely different. We observed instead co-occurrences at the method level, affecting (considering all smells) less than 10% of smelly methods. No single smell combination is outstanding, but Brain Method has the largest share of co-occurrences. Since the smell is defined around the concepts of high size and complexity, the chances of being affected by other smells are higher than for the other ones.

In conclusion, among the six smells we have studied, we can assert that respect to our evaluation, the most dangerous smells according to architecture degradation could be Brain Method, God Class, Shotgun Surgery, Dispersed Coupling. Moreover, we observed that the number of Data Class, Brain Method and Dispersed Coupling in the 74 analyzed systems is very high.

Finally, preliminary evaluations have revealed that the proposed views are useful for the comprehension of the problems captured by the detected code smells; we plan the full implementation and experimentation of the views for future work and to implement our visualization framework using Gephi, an open source software for graph visualization and manipulation. Moreover, we would like to investigate statistical measures to quantify the actual presence of identified code smell relations.

References

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

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

[3] R. L. Nord, I. Ozkaya, H. Koziolek, and P. Avgeriou, "Quantifying software architecture quality report on the first international workshop on software architecture metrics," *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 5, pp. 32–34, Sep. 2014.

[4] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Proc. 2010 IEEE Int'l Conf. Softw. Maintenance (ICSM 2010)*. IEEE, Sep. 2010, pp. 1–10.

[5] P. Oliveira, M. Valente, and F. Paim Lima, "Extracting relative thresholds for source code metrics," in *Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE 2014)*. Antwerp, Belgium: IEEE Computer Society, Feb. 2014, pp. 254–263.

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

[7] F. Arcelli Fontana and S. Maggioni, "Metrics and antipatterns for software quality evaluation," in *Proc. 34th IEEE Software Engineering Workshop (SEW 2011)*. Limerick, Ireland: IEEE, Jun. 2011, pp. 48–56.

[8] D. Binkley, N. Gold, M. Harman, Z. Li, K. Mahdavi, and J. Wegener, "Dependence anti patterns," in *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops, 2008*. L'Aquila, Italy: IEEE, Sep. 2008, pp. 25–34.

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

[10] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *Architectures for Adaptive Software Systems*, ser. Lecture Notes in Computer Science, R. Mirandola, I. Gorton, and C. Hofmeister, Eds. Springer Berlin Heidelberg, 2009, vol. 5581, pp. 146–162.

[11] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, "Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems," in *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development (AOSD '12)*. Potsdam, Germany: ACM, Mar. 2012, pp. 167–178.

[12] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Automat. Soft. Engin.*, pp. 1–32, Dec. 2014.

[13] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *Proc. 17th Asia Pacific Software Engineering Conference (APSEC 2010)*. Sydney, Australia: IEEE, December 2010, pp. 336–345.

[14] I. Ozkaya, A. Diaz-Pace, A. Gurfinkel, and S. Chaki, "Using architecturally significant requirements for guiding system evolution," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*. Madrid, Spain: IEEE Computer Society, March 2010, pp. 127–136.

[15] R. Arcoverde, E. Guimarães, I. M. Bertran, A. Garcia, and Y. Cai, "Prioritization of code anomalies based on architecture sensitiveness," in *Proceedings of the 27th Brazilian Symposium on Software Engineering, (SBES 2013)*, Brasilia, Brazil, Oct. 2013, pp. 69–78.

[16] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR 2011)*. Oldenburg, Germany: IEEE, Mar. 2011, pp. 181–190.

[17] F. Arcelli Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," in *Proceedings of the 6th International Workshop on Emerging Trends in Software Metrics (WETSoM 2015)*. Florence, Italy: IEEE, May 2015, co-located with ICSE 2015.

[18] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[19] V. Ferme, "JCodeOdor: A software quality advisor through design flaws detection," Master's thesis, University of Milano-Bicocca, Milano, Italy, Sep. 2013, http://essere.disco.unimib.it/reverse/files/VFermeMsCThesis2013.pdf.

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

[21] B. Pietrzak and B. Walter, "Leveraging code smell detection with inter-smell relations," in *Extreme Programming and Agile Processes in Software Engineering*, ser. Lecture Notes in Computer Science, P. Abrahamsson, M. Marchesi, and G. Succi, Eds. Springer Berlin / Heidelberg, 2006, vol. 4044, pp. 75–84.

[22] S. Jancke and D. Speicher, "Smell Detection in Context," Diploma Thesis, University of Bonn, 2010.

[23] A. Yamashita, B. Anda, and A. Mockus, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, Dec. 2012.

[24] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of bad smell detection and resolution: A new way to save effort," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 220–235, Jan. 2012.

[25] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger, "Codecrawler: an information visualization tool for program comprehension," in *Proc. 27th International Conference on Software Engineering (ICSE '05)*. St. Louis, MO, USA: ACM, 2005, pp. 672–673.

[26] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009)*. Kaiserslautern, Germany: IEEE, Mar. 2009, pp. 255–258.