Exploratory Study on the Landscape of Inter-smell Relations in Industrial and Open Source Systems

Aiko Yamashita*, Marco Zanoni†, Francesca Arcelli Fontana† and Bartosz Walter‡
*Simula Research Laboratory, Mesan AS, Oslo, Norway
Email: aiko@simula.no
†Department of Informatics, Systems and Communication, University of Milano-Bicocca, Milano, Italy
Email: {marco.zanoni,arcelli}@disco.unimib.it
‡Faculty of Computing, Poznań University of Technology, Poznań, Poland
Email: bartosz.walter@cs.put.poznan.pl

***Abstract*—Code smells are indicators of issues with source code that can hinder software evolution. While a great number of studies have focused on the effects of individual code smells on maintainability, recent work has shown that code smells that appear together in the same file (i.e., collocated smells) can interact with each other, leading to various types of maintenance issues and/or to intensified negative effects. Moreover, it has been found that code smell interactions may occur across coupled files (i.e., coupled smells), with comparable negative effects as the interaction of same-file, collocated smells. Different inter-smell relations have been described in previous work, but few studies have validated them empirically. This study investigates further the phenomena of inter-smell relations (both collocated and coupled smell relations), by analyzing one industrial and two open-source systems. We substantiated the relevance of some inter-smells previously reported in the literature and extend further the landscape of known inter-smell relations. We found that tendencies on inter-smell relations become clearer when considering coupled smells in addition to collocated smells. A major finding is that contextual factors such as the domain and environment may play a major role on the presence, co-presence, and coupling between smells, which suggests that such variables should be considered when conducting smell analysis.**

***Index Terms*— code smells; bad smells; inter-smell relations; smell interaction; dependency analysis; software quality.**

I. Introduction

Code smells are indicators of potentially harmful design shortcomings that can cause difficulties to developers during maintenance. These shortcomings can decrease code quality aspects such as understandability and changeability, and can lead to the introduction of faults [1]. However, the overall capacity of code smells for explaining maintenance problems/effort has been shown to be rather low (see for example the work by Sjøberg et al. [2]). While a great number of studies have focused on the effects of individual code smells on maintainability, recent work has shown that inter-smell relations [3] may provide a better insight on potential maintenance issues. Code smells that appear together in the same file (i.e., collocated smells) can interact with each other, causing different types of problems and intensifying them. Moreover, it has been found that code smell interactions that occur across coupled files (i.e., coupled smells), can lead to comparably negative effects as the interaction of same-file, collocated code smells. Thus, we argue that effects of inter-smell relations on software maintainability is a topic that deserves more attention. This position is further supported by the observation made by Yamashita [4] that in some large classes, the maintenance problems were not so much caused by the complexity that followed from the actual size of the class but rather were a result of interaction effects between different code smells that appeared together in that class. This distinction implies that the current approach for code smell detection and analysis, which is mostly based on analyzing individual smells and not the effects of smell interactions, severely limits the capability of code smell analysis to explain or predict maintenance problems. Observations in [4] suggests that we should have only a modest expectation of the explanatory and predictive power of individual code smells in relation to software maintainability. We know only of one empirical study (by Abbes et al. [5]) that reports on the interaction effects between two concrete code smells (i.e., between God Class and God Method). Yamashita and Moonen [6] reported that interaction effects do occur among code smells and between code smells and other design flaws. This implies that the current approach for code smell analysis (i.e., analyzing individual smells and not the effect of their combinations) limits greatly the capability of code smells to explain much of the maintenance problems caused by design flaws. Another limitation of the current approaches for code smell analysis is that couplings amongst files containing code smells are not considered in the analyses. The findings from [6, 7] suggest that interaction effects between code smells distributed across coupled files may have the same consequences from a practical perspective as interaction effects between code smells collocated in the same file. "Coupled smells" are currently ignored due to the fact that code smells are mostly identified and analyzed at the file level. Consequently, we argue that in order to obtain a better understanding of the role of code smells on maintainability, future studies should integrate dependency analysis in their process as to include coupled smell-interactions.

The innovative aspects of this work are three fold: *i)* We involve a combination of both industrial and open source systems, all of them of considerable size and complexity, to identify and analyze inter-smell relations and *ii)* We conduct a study to corroborate some of the relations described by previous work, and *iii)* We incorporate dependency analysis to investigate coupled smells in addition to the previously studied collocated smells.

The remainder of this paper is as follows. Section II presents the theoretical background and related work. Section III describes the study design, including the context of the systems studied, and the types of analysis conducted. Section IV presents the results of the analysis and Section V discusses some of the findings in more detail and describes the threats to the validity of the results. Section VI presents the conclusion and future work.

II. Theoretical background and related work

A. Code smells

'Code smells' is a term coined by Fowler and Beck [1] (i.e., 'Bad smells in the code') for describing symptoms of deeper issues in the code. They informally described and exemplified 22 different code smells, and related them back to well-known violations of different programming and design principles. Unlike code metrics, smells are easier to interpret for quality assurance/improvement efforts, which has attracted the attention of researchers and practitioners in software engineering. However, they do not directly point to the actual flaw and usually require a certain level of interpretation and additional analysis to determine if they are actually problematic or not. Different approaches for detecting code smells have been proposed and are currently in use. Although Fowler has emphasized the subjective nature of code smells (e.g., "no sets of metrics rivals informed human intuition" [1]), research efforts in the last decade have incorporated automated means for smell detection. For example, *detection strategies* [8] use logical combinations of different code metrics with different threshold values to identify code smells (several commercial and open source tools implement this approach, such as inFusion[1] and PMD[2]). Other detection approaches match different code attributes with refactoring opportunities (e.g., JDeodorant[3]), or employ machine-learning algorithms to discover relations between metrics and code smells (e.g. Arcelli et al. [9], Khomh et al. [10]). Some approaches are not based only on the source code, but they consider code evolution via repository mining, such as the work by Palomba et al. [11]. Finally, some approaches [12] use a wider spectrum of data, by incorporating domain-specific information, and probabilities (i.e., design change propagation probability matrix, or DCPP matrix) to detect code smells.

B. Empirical studies on code smells

A systematic literature review reported in [13] indicated that effort in the Software Engineering research community has been placed mainly on investigating how to detect code smells rather than empirically examining their impact on software quality characteristics. The review identified only three empirical studies investigating the impact of code smells on maintenance [14–16]. In the doctoral thesis by Yamashita [4] a review on empirical studies on code smells is presented, with a focus on the empirical effects of code smells on maintenance. Studies included in this review have indicated that certain individual code smells have deterrent effects as the introduction of defects [14–19] larger maintenance effort [5, 20–22], and larger and more frequent changes in the code [23–25]. The review also includes smell dynamics such as smell evolution and longevity [26–28]. The systematic literature review by [13], the report by [4], as well as recent work by Sjøberg et al. [2] and Yamashita [29] suggest that insofar, the overall capacity of code smell analysis to explain or predict maintenance problems is rather modest.

C. Inter-smell relations

While most of the previously described work focus on the impact of *individual* code smells on different maintenance outcomes, we focus our attention on *relationships* among code smells, and the potential implications that smell interactions have on maintenance. Pietrzak et al., [3] described several types of inter-smell relations to support more accurate code smell detection and to understand better the effects caused by interactions between smells. These relations were supported by examples from the Apache Tomcat[4] code base. Some of the reported relations are:

- Data Class, Feature Envy, Large Class (Trans. Support)
- Large Class, Feature Envy (Plain Support)
- Data Class, Feature Envy (Plain Support)
- Data Class, Inappropriate Intimacy (Rejection)
- Data Class,Feature Envy, Inappropriate Intimacy (Aggregated Support)
- Lazy Class, Large Class (Rejection)
- Parallel Inheritance, Shotgun Surgery (Inclusion)

Previous work on inter-smell relations has mostly been limited to conceptual or anecdotic descriptions; and very few of these inter-smell relations have been corroborated empirically. Mäntylä et al., [30] categorized Fowler's code smells into six groups based on the correlation analysis: Bloaters, Object Orientation Abusers, Change Preventers, Dispensables, Encapsulators and Couplers. The study confirmed the existence of several relations, but also questioned other relations suggested by Fowler. Jancke [31] described different relationships (uses/-forwards/used by) between different design patterns and code smells. Moha et al., [32] proposed a taxonomy of smells and described some relations among design smells, for example:

- Blob and (many) Data Class
- Blob and (Large Class and Low Cohesion)

By inspecting and analyzing source code, Lanza and Marinescu [33] classified twelve smells into 3 categories, called design disharmonies: Identity, Collaboration, and Classification. They asserted that the most of design disharmonies do not appear in isolation (i.e., some of them cluster together), and they describe the most common correlations between the disharmonies, in a type of diagram called correlation web. The identified correlations among the disharmonies manifested via <is/has/uses> relations (See Table I).

[1] http://www.intooitus.com/products/infusion [2] http://pmd.sourceforge.net
[3] http://www.jdeodorant.com/
[4] http://tomcat.apache.org

TABLE I
INTER-SMELL RELATIONS DESCRIBED IN [33]

Type	Relation
Is	Feature Envy is Intense Coupling Brain Method is Dispersed Coupling Tradition Breaker is Refused Parent Bequest
Has	Data Class has Shotgun Surgery God Class has Dispersed Coupling & Intense Coupling God Class has Brain Method God Class has Feature Envy Brain Class has Brain Method Brain Class has Dispersed Coupling & Intense Coupling Data Class has Shotgun Surgery Brain Method has Significant Duplication Tradition Breaker has Significant Duplication
Uses	Dispersed Coupling and Intensive Coupling uses Shotgun Surgery Feature Envy uses Data Class

Yamashita and Moonen [7] conducted principal component analysis (PCA) and an observational analysis [6] over four industrial systems and found that many of the large and complex classes contained other smells, as an indirect consequence of size; as for example they found that:

- Classes with God Class can also contain Feature Envy, Shotgun Surgery, and Interface Segregation Principle (i.e., ISP see [34]) Violation
- Classes with high Coupling can involve Feature Envy or ISP Violation or Shotgun Surgery

Moreover, they found that smell interactions occur across coupled files (i.e., coupled smells), and that those can lead to comparably negative effects as the interaction of same-file, collocated code smells.

III. STUDY DESIGN

In our study, we focus on exploring further the phenomena of inter-smell relations in industrial and open source systems, and on corroborating some of the relations suggested by previous work. In addition to the collocated smells analysis, we consider dependencies amongst classes in order to incorporate coupled smells analysis. The reminder of this section is as follows: Section III-A describes the systems analyzed, and Section III-B describes which and how code smells were detected, how the collocated and coupled smells were identified, and what type of analysis was conducted.

A. Systems under study

In the study we analyzed two open source systems and one industrial system. As previous works show [35, 36], negative consequences stemming from the presence of code smells may vary depending on different factors. For example, the domain of the systems is an important factor for determining smell intensity and its impact on several qualitative software characteristics [36]. To avoid bias related to contextual factors (e.g., project owner, development method), we decided to analyze systems developed in 2 different environments: industrial, and open source. The choice of specific systems was guided by the availability of the necessary data.

***Sys 1: Ebehandling - A grant application system*:** Ebehandling is a series of modules used by The Norwegian Research Council (hereafter the NRC) for managing research grant applications. NRC is responsible for handing out close to 1,000,000,000 euros in research grants every year. Their systems process around 6000 applications per year, and they support the following functional areas: application evaluation, statistics and reporting, and production of contracts and other documents. The code base analyzed consists of 11 web applications (based on a mix of RESTful web interface and Spring MVC[5], and following a Service Oriented Architecture) and was originally developed by Mesan AS. Currently the systems consist of 5840 files, from which 5300 are Java, 240 are JavaScript and 300 are Jspx (Java Server Pages). The size of the Java code analyzed is of 601KLOC. The 11 modules analyzed had undergone 40 major releases and around 15 patch-releases since the start of the project in 2009. We analyzed a snapshot of the most recent revision, dated as for June 5th, 2014.

***Sys 2: ElasticSearch - A search/analytics platform*:** ElasticSearch is a search server based on the Lucene project. It provides a distributed, multitenant-capable full-text search engine with a RESTful web interface and schema-free JSON documents. ElasticSearch is developed in Java and is released as open source and has undergone 102 minor releases and 22 major releases since early 2010. Some of the active users of ElasticSearch are: Github, SoundCloud, Deezer, and The Guardian. The version we analyzed was 1.2.1, which counts with 2951 Java files and 253 KLOC.

***Sys 3: Mahout - A machine-learning library*:** Apache Mahout is a project of the Apache Software Foundation[6] project aiming at producing free implementations of distributed or otherwise scalable machine learning algorithms focused primarily in the areas of collaborative filtering, clustering and classification. Many of the implementations use the Apache Hadoop[7]platform. Mahout also provides Java libraries for common math operations (focused on linear algebra and statistics) and supports primitive Java collections. Mahout has undergone 10 releases since May 2010, and the version analyzed (0.7) counts with 99 files, from which 935 are Java and 12 are Scala. Current size of the system is 92KLOC. Some of the active users of Mahout include: Yahoo mail, ResearchGate, Mendeley, AOL, and Foursquare.

B. Detection and analysis of code smells

For the detection of code smells, we employed inFusion, a commercial successor of iPlasma[8], which identifies 24 different code smells based on the *detection strategies* approach mentioned in Section II-A. In our study, fifteen code smells

[5] http://projects.spring.io/spring-framework/
[6] http://www.apache.org/foundation/
[7] http://hadoop.apache.org/
[8] http://loose.upt.ro/reengineering/research/iplasma

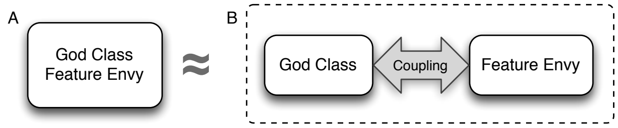

Fig. 1. Example of *Collocated* and *Coupled* Smells

were detected in the systems: Refused Parent Bequest, Distorted Hierarchy, Schizophrenic Class, God Class, Tradition Breaker, Sibling Duplication, Data Clumps, Blob Operation, Feature Envy, Shotgun Surgery, Internal Duplication, Message Chains, External Duplication, Intensive Coupling and Data Class (the definitions are provided in the appendix section in [37], and are also available in the help function of inFusion). The number of detected code smells and the uniformity of the detection approach w.r.t. previous studies guided the choice of the tool. In order to investigate potential inter-smell relations across the systems, we used Principal Component Analysis (PCA), using orthogonal rotation (varimax). First, we conducted PCA for collocated smells, and then we repeated the analysis for coupled smells. We adopted the following definitions of collocated and coupled smells (see Fig. 1):

- *Collocated code smells:* Two or more smells are collocated if they are detected in the same class (Fig. 1-A).
- *Coupled code smells:* Two or more smells are coupled if they are located in artifacts (classes) that are coupled (i.e., they display some type of static dependency (Fig. 1-B).

In this work, we adopted the following definition of a dependency [38]: "... A dependency is when the functioning of one element A requires the presence of another element B..." We interpret the above definition as follows:

1) An outgoing dependency from class X to class Y exists when:
   - X inherits from Y (X is a subclass of Y)
   - X accesses a field of Y
   - X creates a new object of type Y (X instantiates Y)
   - X contains a variable or an attribute of type Y
   - X calls a method declared in type Y
   - X has a method that references Y via return type or parameter
   - X implements B (i.e., when Y is an Interface and X implements the Interface)
2) For each outgoing dependency from X to Y, an incoming dependency from Y to X exists

The tool we used for the dependency analysis was depFinder[9]. We chose the tool due to its capabilities in transforming the resulting graphs. Once we obtained the components from the PCA, we examined the degree of agreement across systems by distinguishing between full-matches (i.e., all variables of a given component in a system have at least one identical counterpart in another system) and partial-matches (i.e., at least two variables in common across two or more components in two or more systems). Subsequently, we examined portions of the code based on a graphical representation of the identified inter-smell by using Gephi[10].

TABLE II
SUMMARY OF THE PCA CONSIDERING COLLOCATED SMELLS

	Ebehandling	**ElasticSearch**	**Mahout**
Variance Explained	60.90%	61.50%	54.60%
Kaiser-Meyer-Olkin	0.48	0.56	0.52
Approx. Chi-Square	2758.95	2250.05	796.66
df	45	105	55
Sig.	.000	.000	.000
RefusedParentBequest		✓	✓
SchizoClass		✓	✓
GodClass	✓	✓	✓
TraditionBreaker		✓	
SiblingDuplication	✓	✓	✓
DataClumps	✓	✓	✓
BlobOperation	✓	✓	✓
FeatureEnvy	✓	✓	✓
ShotgunSurgery		✓	
InternalDuplication	✓	✓	✓
MessageChains	✓	✓	✓
ExternalDuplication	✓	✓	✓
IntensiveCoupling		✓	
DataClass	✓	✓	✓

IV. RESULTS

A. PCA with collocated smells

Table XI displays the percentage of the variance explained by the factors extracted, the sample adequacy measures (Kaiser-Meyer-Olkin, and Bartlett's test of Sphericity) for the PCA for each of the systems, and the smells included in the analysis. Some code smells were excluded since they were not detected in some systems. Field [39] recommends a KMO higher than 0.5 for an acceptable sample, which is the case for all the systems except for Ebehandling (0.48). The Bartlett's test of Sphericity is significant for all systems ($p < .001$), indicating that the correlations between the items are sufficiently large for a satisfactory application of PCA. Despite the KMO for Ebehandling being slightly lower than what recommended, we decided to maintain it, given that the Bartlett's test result was significant for all systems. For Ebehandling, five components had eigenvalues over Kaiser's criterion of 1 and in combination explained 60.9% of the variance. As for ElasticSearch, eight factors where identified, explaining 61.5% of the variance. Finally, for Mahout, five components explaining 54.6% of the variance were extracted. Due to space restrictions, the factor loadings after the rotation for each of the systems are available in a technical report [37].

Fig. 2 displays the resulting components, which shows one common, collocated inter-smell relation across the systems Ebehandling (component 1) and ElasticSearch (component 2): God Class + Feature Envy + Intensive Coupling. This

[9] http://depfinder.sourceforge.net

[10] http://www.gephi.org

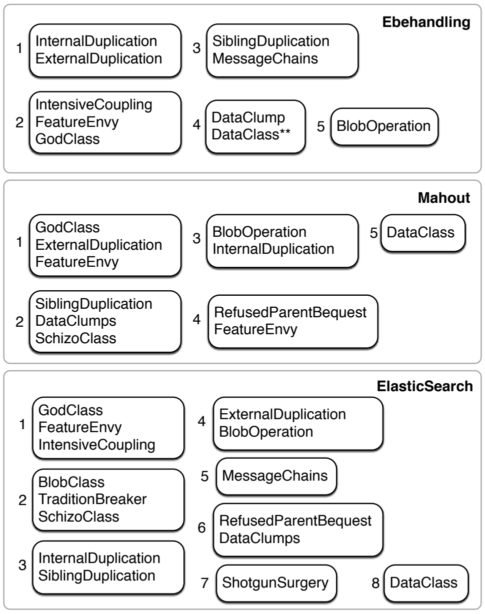

Fig. 2. Components identified for collocated smells

TABLE III
SUMMARY OF THE PCA CONSIDERING COUPLED SMELLS

	Ebehandling			ElasticSearch			Mahout		
Variance Explained	61.76%			56.01%			61.64%		
Kaiser-Meyer-Olkin	0.57			0.72			0.62		
Approx. Chi-Square	23256.05			32755.89			11152.26		
df	496			990			496		
Sig.	.000			.000			.000		
	file	in	out	file	in	out	file	in	out
BlobClass				✓	✓	✓			
RefusedParentBequest				✓	✓	✓	✓	✓	✓
SchizoClass		✓	✓	✓	✓	✓	✓		
GodClass	✓	✓	✓	✓		✓	✓	✓	✓
TraditionBreaker				✓	✓	✓			
SiblingDuplication		✓	✓	✓	✓	✓	✓	✓	✓
DataClumps	✓	✓	✓	✓	✓	✓	✓		✓
BlobOperation	✓	✓	✓	✓	✓	✓	✓	✓	✓
FeatureEnvy		✓	✓	✓	✓	✓	✓	✓	✓
ShotgunSurgery				✓	✓	✓			
InternalDuplication		✓	✓	✓	✓	✓	✓	✓	✓
MessageChains		✓	✓	✓	✓	✓	✓	✓	✓
ExternalDuplication		✓	✓	✓	✓	✓	✓	✓	✓
IntensiveCoupling		✓	✓	✓	✓	✓			
DataClass		✓	✓	✓	✓	✓	✓	✓	✓

further supports the inter-relation of plain support that was suggested by Walter and Pietrzak [3] (i.e., Large Class and Feature Envy), and partially confirms the work by Moha et al., [32] (i.e., Large Class and Low Cohesion). It also confirms the relation supported by Lanza and Marinescu [33], who assert that God Classes often display Dispersed Coupling and Intensive Coupling, that God Classes display Feature Envy, and that Feature Envious methods display Intensive Coupling. These observations are also consistent with the finding by Yamashita and Moonen [7], who observed that God Classes often display Feature Envy, Shotgun Surgery, and ISP Violation (this last design flaw is similar to Dispersed Coupling). In Ebehandling, component 4 contained Data Class is marked with double asterisk (**), since it displayed a negative loading. This indicates that whenever Data Clump is present, Data Class is not present in that file. This observation is quite interesting given that in [7], Data Class and Data Clump were found together in the same component, both with positive loadings. In the other two systems analysed, Data Class appears in its own component, and the negative loading in Ebehandling suggests also that this smell appears alone. Additionally, we could observe some tendencies with respect to smells related to duplication: They tend to cluster in the same files, and often together with smells associated to size and complexity (e.g., Blob Operation, Schizo Class, God Class). See for example components 1, 2 and 3 in Mahout, and component 4 in ElasticSearch.

B. PCA with coupled smells

Table 3 shows the sample adequacy measures. For each of the smells, three dimensions of variables exist: File, In and Out. 'File' indicates that a given smell was found in the file. 'In' means that the file had an incoming dependency from a file that contained a given smell. 'Out' means that the file depends on another file that displays a given smell. All KMO measures indicate that the sample is adequate and Barlett's test of Sphericity is significant. Therefore we can conclude that the sample is good.

Fig. 3–5 display the components identified through PCA. The variables with the prefix 'in_' indicate that there was an incoming dependency from a file that contained a given smell. For example, in Fig. 3, the component 1 would indicate that a file containing Data Class would have incoming dependencies from files that contain Feature Envy, God Class, Data Class or Message Chains.

In Fig. 3–5, the components displaying a full-match are marked in green and the components displaying a partial-match are marked in yellow, where the matching smells are marked in bold. The components that have smells related to duplication (clones) are marked in purple with a light font. In addition, redundant components (i.e. all elements of a component constitute the same smell, but some are marked as 'in_' or 'out_') are marked with dashed edges. One consistent inter-smell relation across Ebehandling and ElasticSearch was: Feature Envy and Data Class (See component 9 in Fig. 3 and component 10 in Fig. 4). This supports the description of transitive support by Walter and Pietrzak, and partially supports work by Moha et al., w.r.t. Blob and (many) Data Classes. Lanza and Marinescu also have discussed that Feature Envy uses Data Classes. Yamashita and Moonen also observed that Feature Envy and Large classes use Data Classes. In addition, we could observe some partial matches identified across systems ('O' for patterns involving outgoing dependencies, 'I' for patterns involving incoming dependencies), as shown in Table IV.

Some redundant components (i.e. a file with a given smell that has incoming and/or outgoing dependencies to files con-

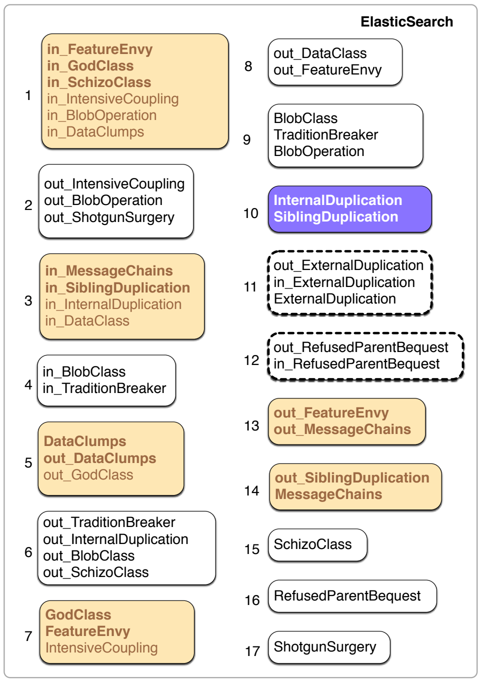

Fig. 4. Components identified in ElasticSearch

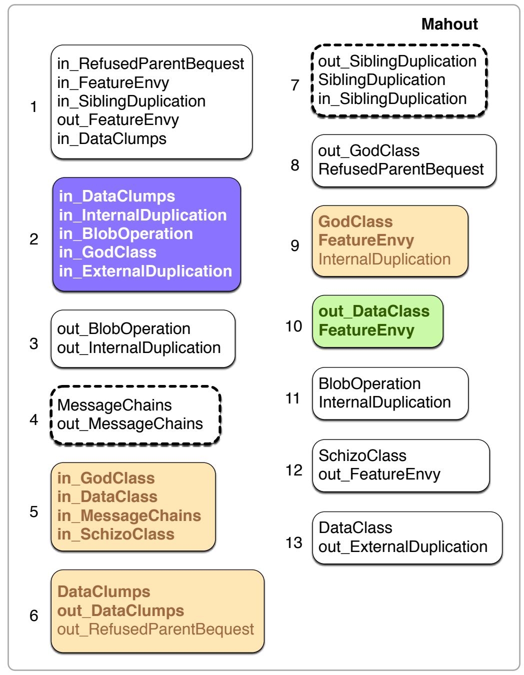

Fig. 5. Components identified in Mahout

taining the exact same smell) were identified for the following smells:

- Data Clumps (Component 8 in Ebehandling)
- External Duplication (Component 11 in ElasticSearch)
- Sibling Duplication (Component 7 in Mahout)
- Blob Operation (Component 3 in Ebehandling)
- Intensive Coupling (Component 10 in Ebehandling)
- Message Chains (Component 4 in Mahout)
- Refused Parent Bequest (Component 12 in ElasticSearch)

This observation suggests that files displaying any of these smells may have a tendency to be coupled with other files with the same smells. This can be useful for achieving higher confidence that a given smell is actually present (e.g., Walter's multi-method approach [40]), or as baseline for estimating the harmfulness level of a smell. If a smell is related to many other smells, this information could be used to prioritize the inspection of artifacts with that specific smell before other ones. For example, if the Brain Method smell co-occurs with the Dispersed Coupling smell and the same Brain Method smell

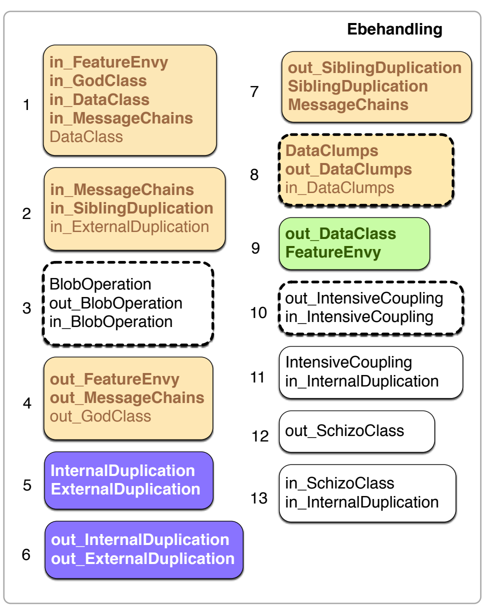

Fig. 3. Components identified in Ebehandling

TABLE IV
INTER-SMELL PATTERNS CONSIDERING COUPLED SMELLS

Name	System (Component)	Smells
Pattern O-1:	Ebehandling (8) Mahout (6) ElasticSearch (5)	Data Clumps Data Clumps in outgoing dependencies
Pattern O-2:	Ebehandling (7) ElasticSearch (14)	Message Chains Sibling Duplication in outgoing dependencies
Pattern O-3:	Ebehandling (4) ElasticSearch (13)	Feature Envy in outgoing dependencies Message Chain in outgoing dependencies
Pattern O-4:	Ebehandling (7) Mahout (7)	Sibling Duplication Sibling Duplication in outgoing dependencies
Pattern I-1:	Ebehandling (1) Mahout (5)	God Class in incoming dependencies Data Class in incoming dependencies Message Chain in incoming dependencies
Pattern I-2:	Ebehandling (1) ElasticSearch (1)	God Class in incoming dependencies Feature Envy in incoming dependencies
Pattern I-3:	Mahout (5) ElasticSearch (1)	God Class in incoming dependencies Schizo Class in incoming dependencies
Pattern I-4:	Ebehandling (2) ElasticSearch (3)	Message Chain in incoming dependencies Sibling Duplication in incoming dependencies

TABLE V
INTER-SMELL PATTERNS AFTER EQUATING COUPLED SMELLS TO COLLOCATED SMELLS

Name	System (Component)	Smells
Pattern 1:	Ebehandling (9) Mahout (10) ElasticSearch (8)	Data Class Feature Envy
Pattern 2:	Ebehandling (7) ElasticSearch (14)	Sibling Duplication Message Chains
Pattern 3:	Mahout (5) ElasticSearch (1)	God Class Schizo Class
Pattern 4:	Ebehandling (1) Mahout (5)	God Class Data Class Message Chains
Pattern 5:	Mahout (9) ElasticSearch (7)	God Class Feature Envy
Pattern 6:	Ebehandling (4) ElasticSearch (13)	Feature Envy Message Chains
Pattern 7:	Ebehandling (5) Mahout (2)	Internal Duplication External Duplication

co-occurs often with the Message Chain smell, one could consider deeming that specific Brain Method as of 'higher risk' for causing maintenance problems. Assuming from a practical perspective, that coupled smells have the same effect as collocated smells (that means, by equating ***in_<smell>*** and ***out_<smell>*** to ***<smell>***), we could identify the following inter-smell relations, where Patterns 1 and 2 are full-matches and the rest are partial-matches (See Table V).

Although there are some modest tendencies across the systems, the topography on the smells across the FLOSS and the industrial systems seems to have its own idiosyncrasy. This further reinforces our view that domain information is extremely important to interpret code metrics for quality purposes. We believe further work should be directed towards better understanding how contextual factors influence the presence and topography of smells in a system.

V. DISCUSSION

In this section, we first analyze some of the extracted patterns in more detail, providing and discussing some examples that were identified via code inspection. Secondly, we discuss the threats to validity of the work.

A. Inspection of inter-smells in the code

We consider some examples of classes presenting the inter-smell patterns we have reported. The first pattern we consider is Pattern O-1 (Data Clumps, out_DataClumps). In this pattern, a class with a smell has an outgoing dependency to another class which also has a smell. In this regards, Pattern O-2 and Pattern O-4 are similar (Pattern O-3 is different because it involves a class without a smell that have dependencies on a class or classes with smells). We chose to report Pattern O-1 because the analyzed systems contained a large number of Data Clumps instances.

In Mahout, most classes involved in this pattern are connected to the `AbstractJob` class. This class serves as superclass for Mahout Hadoop jobs, and defines three `prepareJob` methods with many parameters, all of them detected as Data Clumps. Each algorithm supporting this protocol for specifying jobs has its own subclass of `AbstractJob` and has Data Clumps too. `AbstractJob` redirects its prepareJob methods to the HadoopUtil class, which exposes the same methods with almost the same signature. In this case, the lack of encapsulation of the configuration parameters of a job is propagated to every subclass implementing a job creation protocol.

In Elasticsearch, class `Aggregator` defines a large constructor, containing Data Clumps. This class has many subclasses, and each subclass re-exposes the same creation protocol. Also in this case, the Data Clumps smell is propagated to subclasses. Another artifact containing Data Clumps is the `FieldMapper` interface. It consists of a wide interface with two methods deemed as Data Clumps. Thus, every implementation of this interface contains at least one Data Clumps. Interestingly, only few abstract classes implement the Data Clumps methods. Most of the Data Clumps in the system stem from constructors defined in super classes, and are propagated to subclasses.

In Ebehandling, the relation among Data Clumps is different. In the two open source systems, most classes displaying this pattern were all connected together in a rather large cluster. In this particular system, only few classes were detected as Data Clumps. Many of them are disconnected from other Data Clumps, or only constituted pairs of connected artifacts.

We also wanted to generate a visual aid to guide the code review, thus derived a classification for the classes involved in this pattern, and represented them in Fig. 6. The figure contains three graphs, one for each of the analyzed systems. The graphs represent the classes displaying this particular inter-relation, with the following convention:

- Nodes represent classes, and edges represent dependencies between classes;
- Dark green (*matching*) nodes are the classes matching the pattern, i.e., the "Data Clumps" part of the pattern;
- Light green (*context*) nodes are classes matching the "out_DataClumps" part of the pattern;
- Cyan (*contributor*) nodes are Data Clumps which are not connected to other Data Clumps, i.e., they do not match the pattern;
- Red (*dependency*) nodes are classes with a dependency from/to green nodes;
- The size of the nodes is proportional to the number of outbound dependencies of the respective class;
- The reported edges are the dependencies among classes having Data Clumps, and their dependencies.

When a node belongs to many categories, the following descending priority order is applied to choose the color of the node: *matching*, *context*, *contributor*, *dependency*. As a final note, the *dependency* nodes for ElasticSearch (on the right of the figure) have not been reported, because they are too many to allow reading the graph.

Several observations can be made based on the code inspection and Fig. 6. First, the amount of Data Clumps is very different in the two open source projects and the industrial one. In the industrial system, data encapsulation was taken more into consideration, while the open source systems display a more liberal tendency in those regards. Second, in the open source systems Data Clumps are highly interconnected, while in the industrial system, the opposite occurs. This can be observed in Fig. 6-a, where Mahout appears as one big cluster whiles Ebehandling (Fig. 6-b) displays several disconnected 'islands'. While this can be a consequence of the total number of Data Clumps identified (i.e., the presence of more instances would increase the chances of finding more interconnected instances), it also suggests that in the industrial system, this smell would be less of a risk, as it can be handled (when necessary) by working on small groups of isolated classes. Third, in the two open source systems where the amount of Data Clumps is highest, the main cause of this seems to be the definition of an abstract method or constructor, and its implementation or repetition into all its subclasses. This kind of propagation can be problematic, because if changes are required in the abstract constructor/method, the corresponding Data Clumps must be corrected in all subclasses. In Elasticsearch, classes containing numerous attributes implemented wide interfaces and defined very wide constructors (detected also as Data Clumps). In this way, Data Clumps propagate through apparently unrelated dependencies.

The second pattern we consider is Pattern I-2 (in_GodClass, in_FeatureEnvy). This pattern is very different from the previous one because it suggests that given a class A, if a God Class depends on it, a Feature Envy would depend on class A as well. Similar patterns are Pattern I-1, Pattern I-3, and Pattern I-4. We chose this pattern over others because the relationship among these two smells is well known. Fig. 7 displays the graphs generated for analyzing this pattern. The convention followed for this particular graph differs to the previous one in the following aspects:

- Dark green (*matching*) nodes are classes having God Classes and Feature Envies in their ingoing dependencies;
- Light green (*context*) nodes are classes with God Class and/or Feature Envy that depend on *matching* classes;
- Cyan (*contributor*) nodes are classes with either God Class or Feature Envy, but not dependent on *matching* classes;
- No red (*dependency*) nodes are represented in this graph, because their amount does not allow the comprehension of the graph.

In Mahout (Fig. 7-b), classes matching the pattern are all connected in a single group. Many classes of this group depend on the `Vector` and `Matrix` interfaces and their specializations. Mahout comprises a collection of data mining algorithms, and some of these algorithms depend on a wide variety of abstractions. For example, the `HmmTrainer` class is a God Class and also contains Feature Envious methods. This class alone depends on seven different interfaces, all of them constituting a kind of abstraction, as the previously described `Vector` and `Matrix`. Other ten classes, classified as God Class or containing Feature Envy, depend on overall 14 classes matching this pattern. Only one class (with Feature Envy) was not involved in this pattern, i.e., it did not depend on any class connected also to a God Class. Other God Classes or Feature Envy methods in the system were connected to the basic set of abstraction or utilities described previously.

In Elasticsearch (Fig. 7-c), most classes matching the pattern are connected in a single large group, others in smaller groups, and only three classes that were affected by Feature Envy or God Class did not display any relations. The single large group (similar to the case of Mahout) comprised a set of general-purpose classes and utility classes. Some of these are: `ElasticsearchIllegalArgumentException`, `Settings`, and `Nullable`.

The situation in Ebehandling (Fig. 7-a) is slightly different. There are more isolated Feature Envies/God Classes. Two groups of classes were identified: one large and one small (there is an additional group of classes in cyan that were not considered since they displayed no coupling). The large group displays low cohesiveness, and is divided into three more cohesive subgroups. A first subgroup has to do with DAO (Data Access Object pattern) management classes, and two other classes: `ModelToDaoConverter` and `DaoToModelConverter`. Both classes constitute God

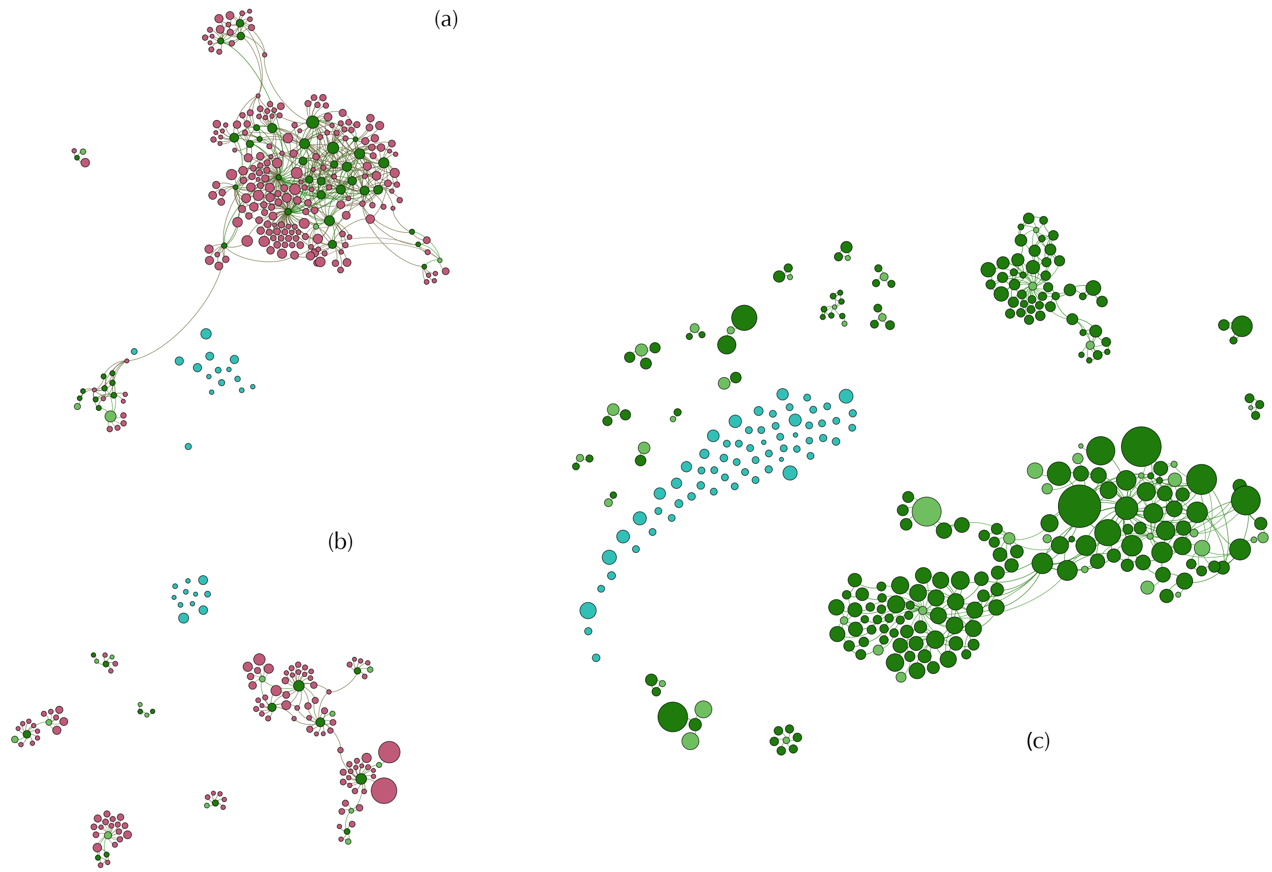

Fig. 6. Pattern O-1 graphs for Mahout (a), Ebehandling (b), and ElasticSearch (c)

Classes, contain Feature Envy methods and use most of the classes matching this pattern. In the second subgroup, the class `AdresselisteIntegrasjonServiceImpl` is both God Class and Feature Envy, and depends on many classes. In the third group, class `Soknad` (God Class) and class `SoknadServiceImpl` (Feature Envy) depend on the same set of classes. A particularity is that all subgroups depend on one single class `DatoHjelper`. Finally, in the small group, seven God Classes and Feature Envy methods depend on a set of 14 other classes.

The results from Ebehandling where discussed with one of the software engineers who had worked on the system. In particular he mentioned that the classes `AdresselisteIntegrasjonServiceImpl`, `Soknad` and `SoknadServiceImpl` have been problematic from a maintenance perspective, but that `DatoHjelper` was not deemed problematic, as it consists of a class with only static methods (thus cannot be instantiated). He noted that the practice of having one 'utility' class containing general purpose methods that can be used by different modules is not rare in these kind of medium-sized information systems. This was also observed in the case of Elasticsearch. These observations warn us against including static classes (like `DatoHjelper`), since they could add noise in the smell analysis. This problem can potentially be handled by using *filters* calibrated to a given context or domain. Filters could help reducing the noise and support better interpretation of code smell-based analysis.

In all the reported examples, many heavily used classes (abstractions, utilities, etc.) matched this pattern, because smelly classes use them. However, many more other non-smelly classes also used the same classes, and this could complicate the interpretation of this pattern. Given the fact that this pattern is recurrent in all the analyzed systems, our interpretation is that this pattern is rooted in a rather typical relation between (collocated) God Class and Feature Envy methods, which was amplified (and thus, captured in the PCA) by the wide dependency set implied by those two smells. However, it is clear that in the open source systems almost all God Classes and Feature Envy methods depend on the same set of classes, with very little exception. We belive that within the open source context, this pattern can potentially be used as an indicator on the level of importance/criticality of the classes used by these two code smells.

B. Threats to Validity

We consider threats to the validity of our study from three perspectives:

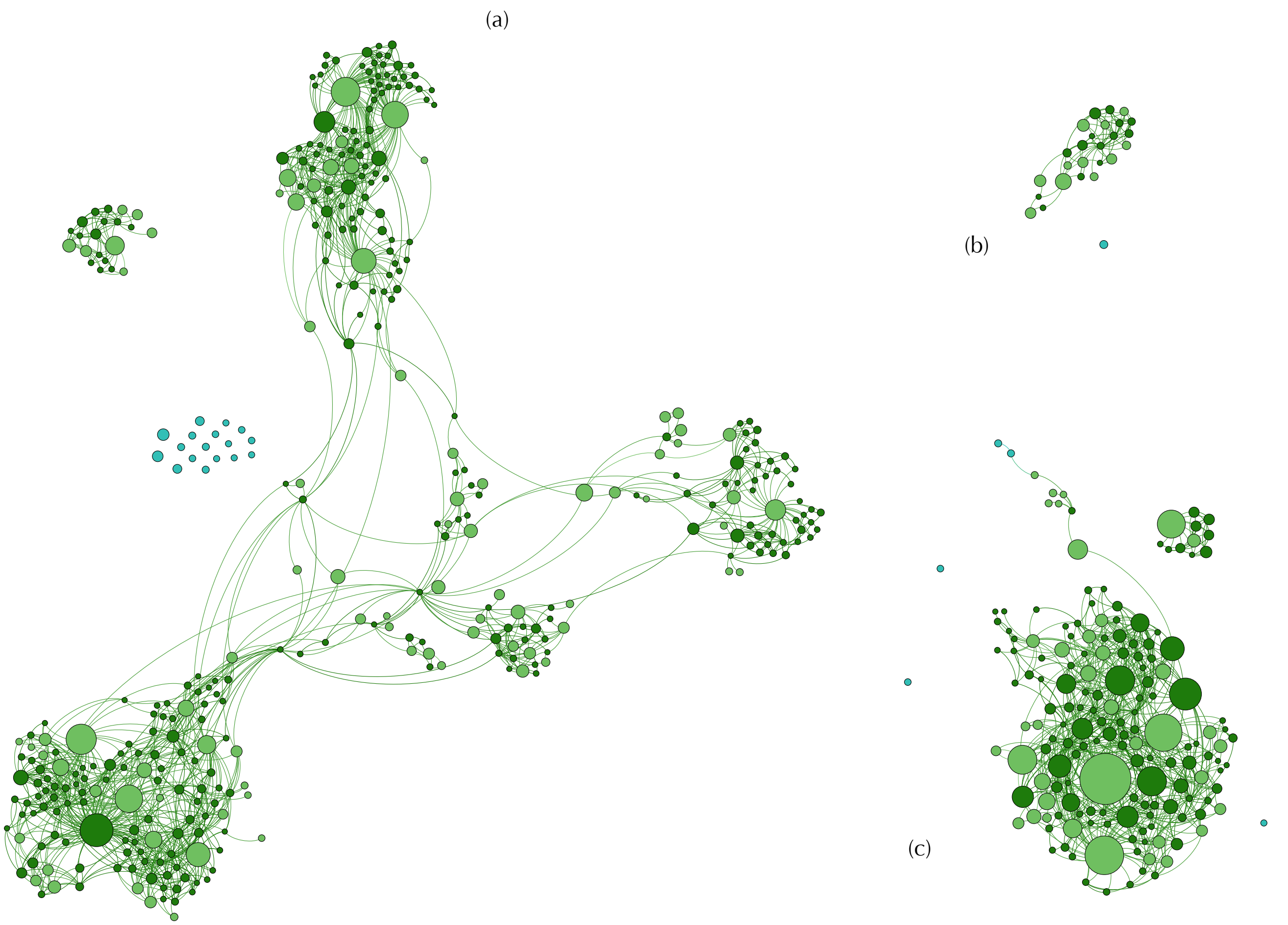

Fig. 7. Pattern I-2 graphs for Ebehandling (a), Mahout (b), and ElasticSearch (c)

1) Construct Validity: The code smells were automatically identified by inFusion to avoid subjective bias. The choice for the detection approach (e.g., threshold values) used by the tool could be a threat to validity. The variations in the detection strategies across different tools might lead to variations in the set of identified smells and consequently, any potential inter-smell relations.

2) Internal Validity: When analyzing the coupled smells (inter-smell relations across coupled classes), we represented the code smells in the dependencies of each class as a single set, i.e., for each class we expressed if a given smell was present or not in any of its dependencies. This representation does not take into account the number of classes a given class is coupled to, nor the number of couplings between each pair of classes. Consequently, some information is lost (e.g., the spread/intensity of the coupling). The patterns inferred from our representation can suffer from a bias (underrepresentation or overrepresentation) given that such information was not considered in the analysis.

3) External Validity: A threat to the external validity of this study is the number of analyzed systems. Although our study confirms some of the previous work on inter-smells, a clear difference in the distribution and interaction of smells could be observed between the open source systems and the industrial one. This further reinforces our view that domain information is extremely important to interpret code metrics for quality purposes. We believe further work should be directed towards better understanding how contextual factors influence the presence and topography of code smells in a system.

VI. Conclusion and Future Work

This paper reports on an empirical study that explores further inter-smell relations in two open source systems and one industrial system, all of them with considerable size and development history. Our study confirms some of the previous conjectures and empirical work on inter-smell relations, and constitutes the first study considering dependencies for analyzing inter-smells. Our results provide: 1) empirical evidence to guide the focus on some of the previously known inter-smells, and 2) an outline on emergent inter-smell relations that can be investigated further as to assess if they can constitute more accurate indicators of maintainability issues. Furthermore, results from the PCA and the code inspection revealed idiosyncratic differences between the two open source and the industrial system, suggesting that contextual factors may play a major role on the landscape of inter-smells. Future work will consist of investigating whether explanatory models (i.e., for explaining defect incidence) considering inter-smells perform better than models only considering individual code smells. We plan to build a logistic regression model using as dependent variable the presence of defects in a software artifact. We plan to create a model only considering individual effects of code smells, and then a model considering interaction effects (between collocated and coupled smells), and we will compare the performance (fit) between the initial (individually-based) and the second (interaction-based) models.

***Acknowledgments*:** The authors thank Intooitus for providing the license for inFusion, and Øystein Skadsem for his support during the data collection and for insightful discussions.

References

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

[2] D. I. Sjoberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dyba, "Quantifying the Effect of Code Smells on Maintenance Effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2013.

[3] B. Pietrzak and B. Walter, "Leveraging Code Smell Detection with Inter-smell relations," in *Extreme Programming and Agile Processes in Softw. Eng. (XP)*. Springer Berlin / Heidelberg, 2006, pp. 75–84.

[4] A. Yamashita, "Assessing the Capability of Code Smells to Support Software Maintainability Assessments: Empirical Inquiry and Methodological Approach," Doctoral Thesis, University of Oslo, 2012.

[5] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension," in *15th European Conf. Softw. Maintenance and ReEng.* IEEE, 2011, pp. 181–190.

[6] A. Yamashita and L. Moonen, "To what extent can maintenance problems be predicted by code smell detection? An empirical study," *Information and Software Technology*, vol. 55, no. 12, pp. 2223–2242, 2013.

[7] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *2013 35th Int'l Conf. Softw. Eng.* IEEE, 2013, pp. 682–691.

[8] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9:1–9:13, 2012.

[9] F. Arcelli Fontana, M. Zanoni, A. Marino, and M. V. Mantyla, "Code Smell Detection: Towards a Machine Learning-Based Approach," in *2013 IEEE Int'l Conf. Softw. Maintenance*. IEEE, 2013, pp. 396–399.

[10] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "BDTEX: A GQM-based Bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.

[11] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *2013 28th IEEE/ACM Int'l Conf. Automated Softw. Eng.* IEEE, 2013, pp. 268–278.

[12] A. A. Rao and K. N. Reddy, "Detecting bad smells in object oriented design using design change propagation probability matrix," in *Int'l MultiConf. Engineers and Computer Scientists*, 2008, pp. 1001–1007.

[13] M. Zhang, T. Hall, and N. Baddoo, "Code Bad Smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 3, pp. 179–202, 2011.

[14] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *IEEE Symposium on Softw. Metrics*, 2002, pp. 87–94.

[15] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, vol. 80, no. 7, pp. 1120–1128, 2007.

[16] C. Kapser and M. Godfrey, "'Cloning Considered Harmful' Considered Harmful," in *Working Conf. Reverse Eng.*, ser. WCRE '06. Washington, DC, USA: IEEE, 2006, pp. 19–28.

[17] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Int'l Conf. Softw. Eng.*, 2009, pp. 485–495.

[18] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the Impact of Design Flaws on Software Defects," in *Int'l Conf. Quality Softw.*, 2010, pp. 23–31.

[19] F. Rahman, C. Bird, and P. Devanbu, "Clones: what is that smell?" *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 503–530, 2011.

[20] I. Deligiannis, M. Shepperd, M. Roumeliotis, and I. Stamelos, "An empirical investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 65, no. 2, pp. 127–139, 2003.

[21] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd, "A controlled experiment investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 72, no. 2, pp. 129–143, 2004.

[22] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *IEEE Int'l Conf. Softw. Maintenance*, 2008, pp. 227–236.

[23] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An Exploratory Study of the Impact of Code Smells on Software Change-proneness," in *Working Conf. Reverse Eng.* IEEE, 2009, pp. 75–84.

[24] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," in *Joint 10th European Software Engineering Conference (ESEC) and 13th ACM SIGSOFT Symposium on the Foundations of Softw. Eng. (FSE-13)*, 2005, pp. 187–196.

[25] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *3rd Int'l Symposium on Empirical Softw. Eng. and Measurement (ESEM)*. IEEE, 2009, pp. 390–400.

[26] A. Chatzigeorgiou and A. Manakos, "Investigating the Evolution of Bad Smells in Object-Oriented Code," in *2010 Seventh Int'l Conf. the Quality of Information and Communications Technology*. IEEE, 2010, pp. 106–115.

[27] R. Peters and A. Zaidman, "Evaluating the Lifespan of Code Smells using Software Repository Mining," in *2012 16th European Conf. Softw. Maintenance and ReEng.* IEEE, 2012, pp. 411–416.

[28] N. Göde and J. Harder, "Oops! . . . I changed it again," in *Proceeding of the 5th Int'l Ws. Softw. clones - IWSC '11*. New York, New York, USA: ACM Press, 2011, p. 14.

[29] A. Yamashita, "Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data," *Empirical Software Engineering*, vol. 19, no. 4, pp. 1111–1143, 2013.

[30] M. V. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *IEEE Int'l Conf. Softw. Maintenance*, 2003, pp. 381–384.

[31] S. Jancke, "Smell Detection in Context," Diplomarbeit, Rheinische Friedrich-Wilhelms-Universität Bonn, 2010.

[32] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[33] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Publishing Company, Inc., 2005.

[34] R. C. Martin, *Agile Software Development, Principles, Patterns and Practice*. Prentice Hall, 2002.

[35] Y. Guo, C. Seaman, N. Zazworka, and F. Shull, "Domain-specific tailoring of code smells," in *Proceedings of the 32nd ACM/IEEE Int'l Conf. Softw. Eng. - ICSE '10*, vol. 2. New York, New York, USA: ACM Press, 2010, p. 167.

[36] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11, no. 3, pp. 395–431, 2006.

[37] A. Yamashita, B. Walter, F. Arcelli, and M. Zanoni, "Exploring the landscape of inter-smell relations in industrial and open source systems. Technical Report. No. 2014-15," Simula Research Laboratory, Oslo, Tech. Rep., 2014.

[38] W. Stevens and G. Myers, "Structured Design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[39] A. Field and J. Miles, *Discovering Statistics Using SAS*. SAGE, 2011.

[40] B. Walter and B. Pietrzak, "Multi-criteria Detection of Bad Smells in Code with UTA Method," in *Extreme Programming and Agile Processes in Softw. Eng. (XP)*, ser. Lecture Notes in Computer Science, H. Baumeister, M. Marchesi, and M. Holcombe, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3556, pp. 154–161.

Appendix

A.1. Definition and Detection Strategy of Analyzed Code Smells (extracted from InFusion)

Refused Parent Bequest: Refused Parent Bequest is a design flaw affecting subclasses in an inheritance hierarchy. The relation between a parent class and its children is intended to be an intimate one, more special than the collaboration between two unrelated classes. This special collaboration is based on a category of members (methods and data) especially designated by the base class to be used by its descendants, namely the protected members. But if a child class refuses to use this special bequest prepared by its parent, then this is a sign that something is wrong within that inheritance relation. Extending base classes without looking at what they have to offer introduces duplication and leads to class interfaces that become incoherent and non-cohesive.

Detection rule – The detection rule for this design flaw looks for classes that either do not use or specialize inherited members, or, in the case of C++, classes that use "implementation inheritance" (i.e. private or protected inheritance).

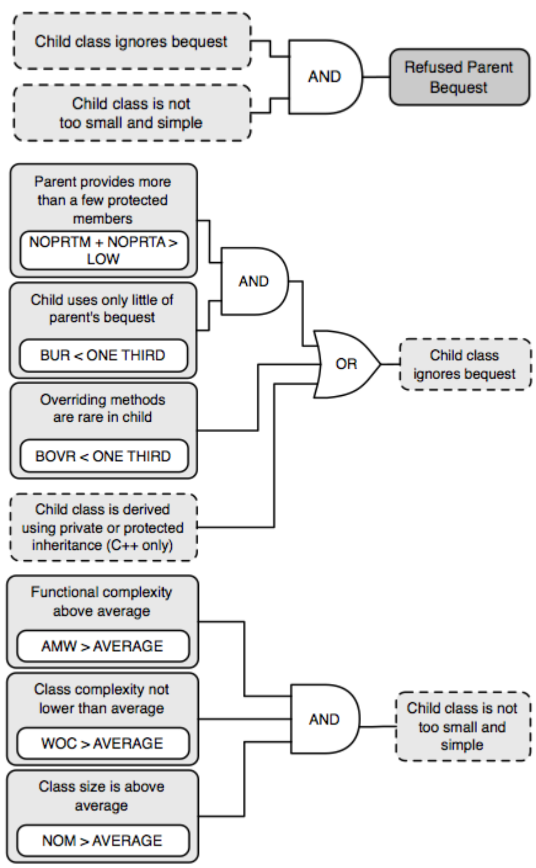

Fig. 8. Refused Parent Detection Strategy

Metrics used – AMW, BOVR, BUR, NOM, NOPRTM, WOC

Literature references – [Riel96] A.J. Riel. Object-Oriented Design Heuristics. Addison-Wesley, 1996. [Lanza06] Michele Lanza and Radu Marinescu. Object-Oriented Metrics in Practice. Springer, 2006. [Martin07] Robert C. Martin. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, 2007.

Distorted Hierarchy: A Distorted Hierarchy is an inheritance hierarchy that is unusually narrow and deep. This design flaw is inspired by one of Arthur Riel's heuristics, which says that "in practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory. A popular value for this depth is six". Having an inheritance hierarchy that is too deep may cause maintainers "to get lost" in the hierarchy making the system in general harder to maintain. The presence of a Distorted Hierarchy not only has a negative influence on the complexity of the code, but also it is a sign of a potential encapsulation problem, in the sense that encapsulation occurs at a too fine-grained level.

Detection rule – The detection rule employs several metrics that measure the depth and width of an inheritance hierarchy. The rule is applied to a class, and checks the hierarchy which originates from that class.

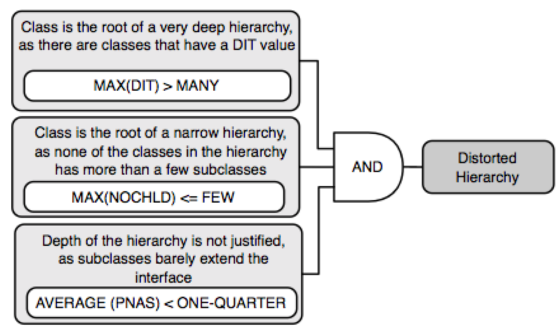

Fig. 9. Distorted Hierarchy Detection Strategy

Metrics used – DIT, NOCHLD, PNAS

Literature references – [Riel96] A.J. Riel. Object-Oriented Design Heuristics. Addison-Wesley, 1996.

Schizophrenic Class: In object oriented design, a class should not capture more than one key abstraction. Key abstractions are defined as the main entities within a domain model, and often show up as nouns within requirements specifications [Riel96]. A key entity is an abstraction that stands on its own in the abstract model that results from modeling activities. A "schizophrenic class" is a class that captures two or more key abstractions. It negatively affects the ability to understand and change in isolation the individual abstractions that it captures.

Detection rule – The detection rule looks for classes that have a very low cohesion, and define large interfaces that are used by disjoint groups of clients.

Metrics used – NOPUBM, TCC

Literature references – [Riel96] A.J. Riel. Object-Oriented Design Heuristics. Addison-Wesley, 1996. [Martin07] Robert

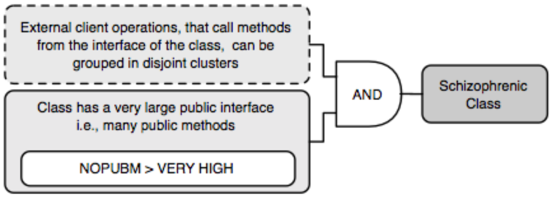

Fig. 10. Schizophrenic Class Detection Strategy

C. Martin. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, 2007.

God Class: A "god class" is a class that manipulates data that belongs to other classes in the system. It is a clear case of braking the encapsulation of the foreign data providers. The "god class" tends to concentrate functionality from several unrelated classes, while at the same time increasing coupling in the system. The god class itself is probably not very cohesive and because of its size and inherent complexity it will have a clear negative impact on the maintainability of the system.

Detection rule – The detection rule employs coupling, cohesion and complexity metrics in order to filter out those complex, non-cohesive classes that access data from more than a few unrelated providers.

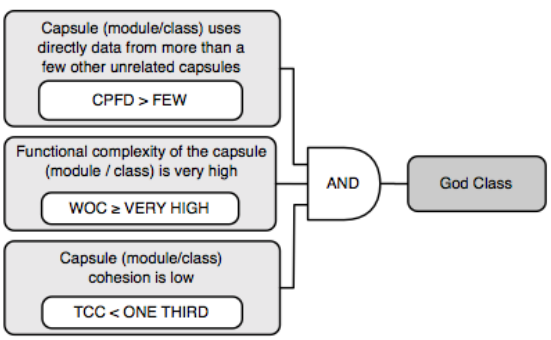

Fig. 11. God Class Detection Strategy

Metrics used – CPFD, TCC, WOC

Literature references – [Riel96] A.J. Riel. Object-Oriented Design Heuristics. Addison-Wesley, 1996. [Fowler99] Martin Fowler. Refactoring. Improving the Design of Existing Code. Addison- Wesley, 1999. [Lanza06] Michele Lanza and Radu Marinescu. Object-Oriented Metrics in Practice. Springer, 2006.

Tradition Breaker: This design flaw takes its name from the principle that the interface of a class (i.e., the services that it provides to the rest of the system) should increase in an evolutionary fashion, and not deny the interface defined by the base classes. This means that a derived class should not break the inherited "tradition", by denying any services (public methods) defined by the base classes. How can a derived class deny a service? In C++ this can be done two ways: (1)

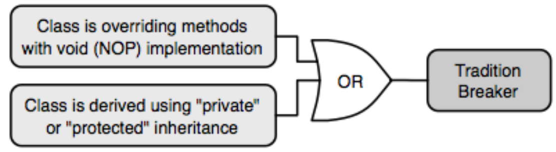

Fig. 12. Tradition Breaker Detection Strategy

by replacing the inherited implementation with a NOP (No-OPeration, empty) method body; (2) by using "private" or "protected" inheritance, which means that the derived class is reducing the visibility of the services "published" by the base classes. In other words, it changes (more exactly cuts a part of) the contract that the base classes have defined. This is a sign that something is wrong either with the definition of the child's class interface or with its classification relation.

Detection rule – The detection rule for "tradition breaker" filters out those subclasses that do one of the two things described above (method "NOP-ing" and/or use a non-public inheritance).

Literature references – [Riel96] A.J. Riel. Object-Oriented Design Heuristics. Addison-Wesley, 1996. [Lanza06] Michele Lanza and Radu Marinescu. Object-Oriented Metrics in Practice. Springer, 2006.

Sibling Duplication: Sibling Duplication means duplication between siblings in an inheritance hierarchy. Code duplication harms the uniqueness of entities within a system. Two or more siblings that define a similar functionality make it much harder to locate errors because the assumption "only class X implements this, therefore the error can be found there" does not hold anymore. Thus, the presence of code duplication has (at least) a double negative impact on the quality of a system: (1) the bloating of the system and (2) the co-evolution of clones (the clones do not all evolve the same way) which also implies the cloning of errors.

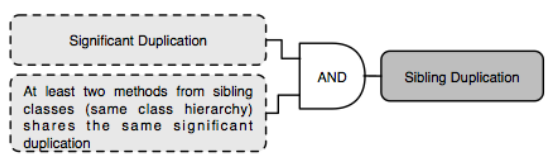

Fig. 13. Sibling Duplication Detection Strategy

Detection rule – The detection of code duplication plays an essential role in the assessment and improvement of a design. But detected clones might not be relevant if they are too small or if they are analyzed in isolation. In this context, the goal of this detection rule is to capture those portions of code that contain a significant amount of duplication.

The detection rules is based on the following metrics:

- LB (Line Bias), measuring the distance between two consecutive exact clones, i.e., the number of non-matching lines of code between two exact clones.

- SDC (Size of Duplication Chain), measuring the total size of a duplication chain in terms of lines of code. A duplication chain may contain one or more exactly cloned fragments, potentially interrupted by small fragments of non-identical code. The size of the duplication chain is the most relevant metric, because it measures the area of code that contains code duplication.
- SEC (Size of Exact Clone), measuring the size of a clone in terms of lines of code. The size of a clone is relevant, because in most of the cases our interest in a piece of duplicated code is proportional to its size.

Metrics used – LB, LOC (indirectly), SDC, SEC

Literature references – [Brown98] W.J. Brown, R.C. Malveau, W.H. Brown, H.W. McCormick, and T.J. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, 1998. [Fowler99] Martin Fowler. Refactoring. Improving the Design of Existing Code. Addison- Wesley, 1999. [Hunt00] Andrew Hunt and David Thomas. The Pragmatic Programmer: From Journeyman to Master. Addison Wesley, 2000.

Data Clumps: Data Clumps is a design flaw inspired by Fowler. They represent groups of data that appear together over and over again, as parameters that are passed to operations throughout the system. Data clumps are good candidates to become objects (in the case of object oriented code) and structure types (in the case of procedural code). They represent cases of bad/lacking encapsulation and have a negative contribution on the ease of maintaining those parts of the system that use the data clumps.

Detection rule – Data clumps are detected by looking for duplicate sets of formal parameter identifiers of a significant size, in methods that tend to have a high number of parameters.

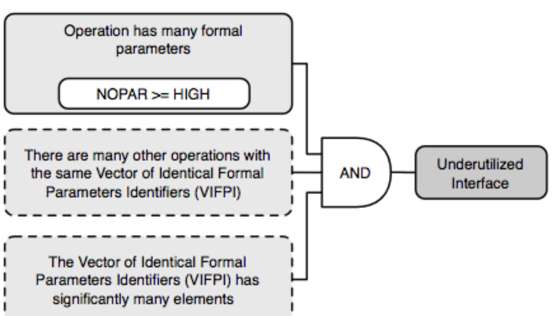

Fig. 14. Data Clumps Detection Strategy

Metrics used – NOPAR

Literature references – [Fowler99] Martin Fowler. Refactoring. Improving the Design of Existing Code. Addison- Wesley, 1999.

Blob Operation: A Blob Operation is a very large and complex operation, which tends to centralize too much of the functionality of a class or module. Such an operation usually starts normal and grows over time until it gets out of control, becoming hard to read and maintain.

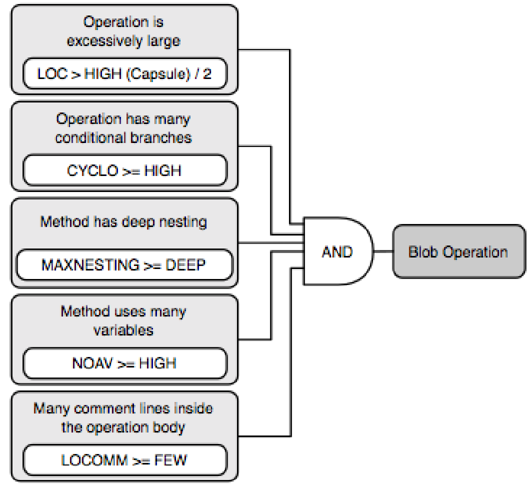

Fig. 15. Blob Operation Detection Strategy

Detection rule – Blob Operation are detected using a combination of size and complexity metrics, as show below.

Metrics used – CYCLO, LOC, LOCOMM, MAXNESTING, NOAV

Literature references – [Brown98] W.J. Brown, R.C. Malveau, W.H. Brown, H.W. McCormick, and T.J. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, 1998. [Fowler99] Martin Fowler. Refactoring. Improving the Design of Existing Code. Addison- Wesley, 1999. [Lanza06] Michele Lanza and Radu Marinescu. Object-Oriented Metrics in Practice. Springer, 2006.

Feature Envy: Classes and modules are mechanisms for keeping together data and the operations that process that data. The Feature Envy design flaw refers to functions or methods that seem more interested in the data of other capsules than the data of those in which they reside. These “envious operations” access either directly or via accessor methods (in the object oriented world) a lot of data that belong to other capsules. This situation is a string indication that the affected method was probably misplaced and that it should be moved to the capsule that defines the “envied data”.

Detection rule – The detection rule closely follows that of Lanza06, by looking for operations that access many data from a few foreign capsules.

Metrics used – ATFD, FDP, LDA

Literature references – [Riel96] A.J. Riel. Object-Oriented Design Heuristics. Addison-Wesley, 1996. [Fowler99] Martin Fowler. Refactoring. Improving the Design of Existing Code. Addison- Wesley, 1999. [Lanza06] Michele Lanza and Radu Marinescu. Object-Oriented Metrics in Practice. Springer, 2006.

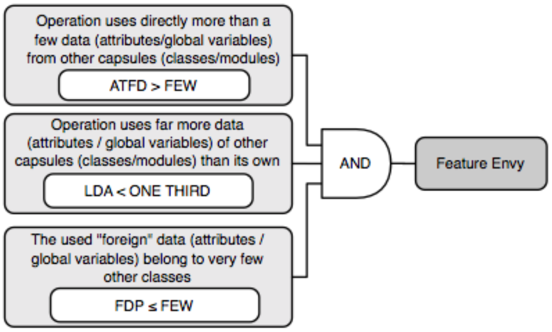

Fig. 16. Feature Envy Detection Strategy

Shotgun Surgery: Shotgun Surgery is a design flaw affecting an operation, whose change implies many changes to a lot of different operations in various classes or modules. This particular flaw tackles the issue of strong afferent (incoming) coupling and it regards not only the coupling strength but also the coupling dispersion. It refers to operations having significant reasons to change (high level of efferent coupling) and whose changes are likely to have a huge impact on the system. These operations are also known as "bottleneck operations" as they are both called by many operations dispersed over many places in the system, and at the same time they depend on many other operations. If a change occurs in such an operation myriads of other methods and functions in various places might need to change as well. As a result, it is easy to miss a required change, thus causing maintenance problems.

Detection rule – The detection rule for Shotgun Surgery uses a combination of coupling, size and complexity metrics, as shown below.

Metrics used – CYCLO, ICDO, ICIO, LOC, MAXNESTING, OCDO, OCIO

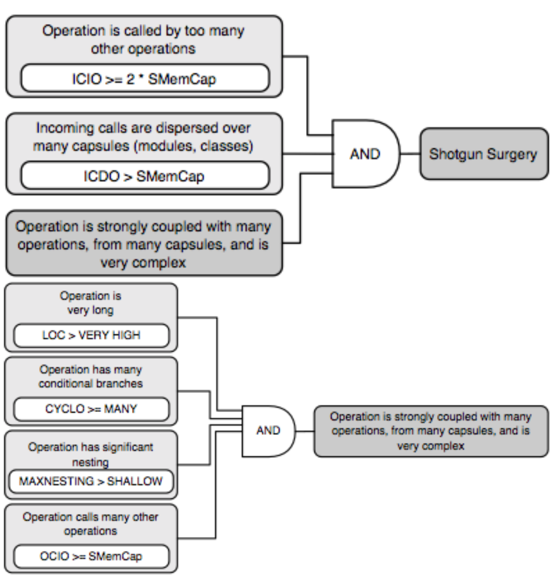

Fig. 17. Shotgun Surgery Detection Strategy

Literature references – [Fowler99] Martin Fowler. Refactoring. Improving the Design of Existing Code. Addison- Wesley, 1999. [Lanza06] Michele Lanza and Radu Marinescu. Object-Oriented Metrics in Practice. Springer, 2006.

Internal Duplication: Internal Duplication means duplication between portions of the same class or module. Code duplication harms the uniqueness of entities within a system. For example, an operation that offers a certain functionality should be solely responsible for that functionality. If duplication appears, it becomes much harder to locate errors because the assumption "only operation X implements this, therefore the error can be found there" does not hold anymore. Thus, the presence of code duplication has (at least) a double negative impact on the quality of a capsule: (1) the bloating of the class or module and (2) the co-evolution of clones (the clones do not all evolve the same way) which also implies the cloning of errors.

Detection rule – The detection of code duplication plays an essential role in the assessment and improvement of a design. But detected clones might not be relevant if they are too small or if they are analyzed in isolation. In this context, the goal of this detection rule is to capture those portions of code that contain a significant amount of duplication. The detection rules is based on the following metrics:

- LB (Line Bias), measuring the distance between two consecutive exact clones, i.e., the number of non-matching lines of code between two exact clones.
- SDC (Size of Duplication Chain), measuring the total size of a duplication chain in terms of lines of code. A duplication chain may contain one or more exactly cloned fragments, potentially interrupted by small fragments of non-identical code. The size of the duplication chain is the most relevant metric, because it measures the area of code that contains code duplication.
- SEC (Size of Exact Clone), measuring the size of a clone in terms of lines of code. The size of a clone is relevant, because in most of the cases our interest in a piece of duplicated code is proportional to its size.

Metrics used – LB, LOC (indirectly), SDC, SEC

Literature references – [Brown98] W.J. Brown, R.C. Malveau, W.H. Brown, H.W. McCormick, and T.J. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, 1998. [Fowler99] Martin Fowler. Refactoring. Improving the Design of Existing Code.

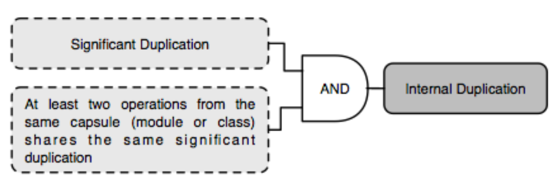

Fig. 18. Internal Duplication Detection Strategy

Addison- Wesley, 1999. [Hunt00] Andrew Hunt and David Thomas. The Pragmatic Programmer: From Journeyman to Master. Addison Wesley, 2000.

Message Chains: This design flaw corresponds to the situation in which a method calls many data exposer methods that belong to other classes (i.e. including also accessor methods, but not limited to these, as data exposers can be also static methods, that return an object that is part of that class). The typical way in which this design flaw manifests itself is when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another object, and so on. You may see these as a long line of getter-like methods, or as a sequence of temporary variables. Navigating this way means the client is coupled to the structure of the navigation. Any change to the intermediate relationships causes the client to have to change.

Detection rule – The detection rule looks for methods that call a significant number of data exposer methods that belong to more than a few classes. Additionally, the detection rule checks if the return types of the various data exposer methods can be connected in a "chain" of navigation, which is typical for cases where the "Law of Demeter" is violated. The longer this navigational chain the more significant the problem is.

Metrics used – OCIO, ODD

Literature references – [Brown98] W.J. Brown, R.C. Malveau, W.H. Brown, H.W. McCormick, and T.J. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, 1998. [Fowler99] Martin Fowler. Refactoring. Improving the Design of Existing Code. Addison- Wesley, 1999. [Lanza06] Michele Lanza and Radu Marinescu. Object-Oriented Metrics in Practice. Springer, 2006.

External Duplication: External Duplication means duplication between unrelated capsules of the system. Code duplication harms the uniqueness of entities within a system. For example, a class that offers a certain functionality should be solely responsible for that functionality. If duplication appears, it becomes much harder to locate errors because the assumption "only class X implements this, therefore the error can be found there" does not hold anymore. Thus, the presence of code duplication has (at least) a double negative impact on the quality of a system: (1) the bloating of the system and (2) the co-evolution of clones (the clones do not all evolve the same way) which also implies the cloning of errors.

Detection rule – The detection of code duplication plays an essential role in the assessment and improvement of a design. But detected clones might not be relevant if they are too small or if they are analyzed in isolation. In this context, the goal of this detection rule is to capture those portions of code that contain a significant amount of duplication. The detection rules is based on the following metrics:

- LB (Line Bias), measuring the distance between two consecutive exact clones, i.e., the number of non-matching lines of code between two exact clones.
- SDC (Size of Duplication Chain), measuring the total size of a duplication chain in terms of lines of code. A duplication chain may contain one or more exactly cloned fragments, potentially interrupted by small fragments of non-identical code. The size of the duplication chain is the most relevant metric, because it measures the area of code that contains code duplication.
- SEC (Size of Exact Clone), measuring the size of a clone in terms of lines of code. The size of a clone is relevant, because in most of the cases our interest in a piece of duplicated code is proportional to its size.

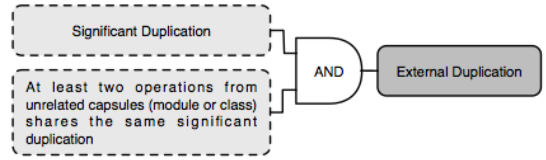

Fig. 19. External Duplication Detection Strategy

Metrics used – LB, LOC (indirectly), SDC, SEC

Literature references – [Brown98] W.J. Brown, R.C. Malveau, W.H. Brown, H.W. McCormick, and T.J. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, 1998. [Fowler99] Martin Fowler. Refactoring. Improving the Design of Existing Code. Addison- Wesley, 1999. [Hunt00] Andrew Hunt and David Thomas. The Pragmatic Programmer: From Journeyman to Master. Addison Wesley, 2000.

Intensive Coupling: Intensive Coupling if a design flaw that affects operations, which are tied to too many other operations in the system, whereby these provider operations are dispersed only into one or a few capsules. An operation which is intensively coupled to operations from a handful of other capsules is strongly bound to those capsules. Oftentimes, intensive coupling points to a more subtle problem (i.e. the capsules providing the many operations invoked by the intensely coupled operation do not provide a service at the appropraite abstraction level. Consequently, understanding the relation between the two sides becomes more difficult.

Detection rule – The detection strategy is based on two main conditions that must be fulfilled simultaneously: the operation must invoke many operations from few foreign capsules and its MAXNESTING metric must be low.

Metrics used – DR, MAXNESTING, OCIO

Literature references – [Riel96] A.J. Riel. Object-Oriented Design Heuristics. Addison-Wesley, 1996. [Fowler99] Martin Fowler. Refactoring. Improving the Design of Existing Code. Addison- Wesley, 1999. [Lanza06] Michele Lanza and Radu Marinescu. Object-Oriented Metrics in Practice. Springer, 2006.

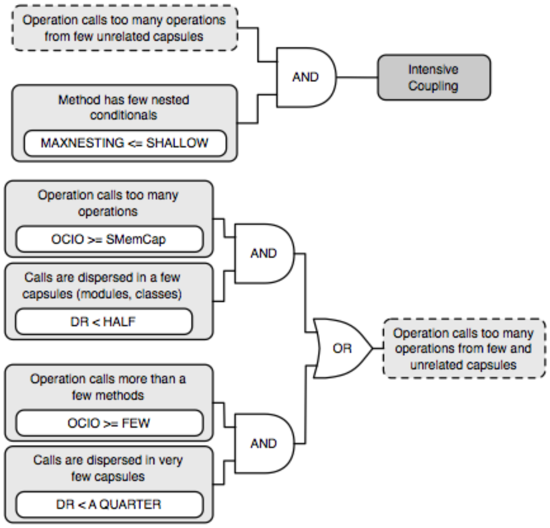

Fig. 20. Intensive Coupling Detection Strategy

Data Class: Data Classes are "dumb" data holders, without complex functionality, but which are usually heavily relied upon by other classes in the system. The lack of functionally relevant methods may indicate that related data and behavior are not kept in one place: this is a sign of a non-object-oriented design. Data classes are the manifestation of a lacking encapsulation of data, and of a poor data-functionality proximity. By allowing other modules or classes to access their internal data, data classes contribute to a brittle, and harder to maintain design.

Detection rule – Data Classes are detected by searching for "lightweight" classes, i.e., classes which provide almost no functionality through their interfaces, but define many accessors (get/set methods) or declare many public attributes.

Metrics used – CW, NOACCM, NOPUBA, WOC

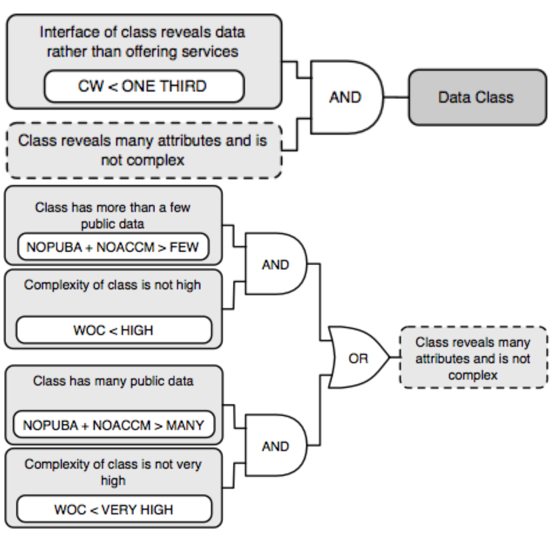

Fig. 21. Data Class Detection Strategy

Literature references – [Riel96] A.J. Riel. Object-Oriented Design Heuristics. Addison-Wesley, 1996. [Fowler99] Martin Fowler. Refactoring. Improving the Design of Existing Code. Addison- Wesley, 1999. [Lanza06] Michele Lanza and Radu Marinescu. Object-Oriented Metrics in Practice. Springer, 2006.

A.2. Factor loadings for the Principal Component Analysis

TABLE VI
COLLOCATED FACTOR LOADINGS AFTER ROTATION (EBEHANDLING)

Ebehandling Rotated Component Matrix	Component				
	1	2	3	4	5
InternalDuplication	,910				
ExternalDuplication	,889				
IntensiveCoupling		,714			
FeatureEnvy		,642			
GodClass		,478			
SiblingDuplication			,706		
MessageChains			,673		
DataClumps				,812	
DataClass				-,563	
BlobOperation					,941

TABLE VII
COLLOCATED FACTOR LOADINGS AFTER ROTATION (MAHOUT)

Mahout Rotated Component Matrix	Component				
	1	2	3	4	5
GodClass	,796				
ExternalDuplication	,514				
SiblingDuplication		,743			
DataClumps		,689			
SchizoClass		,484			
BlobOperation			,773		
InternalDuplication			,728		
RefusedParentBequest				,682	
FeatureEnvy	,462			,625	
DataClass					,879
MessageChains					

TABLE VIII
COLLOCATED FACTOR LOADINGS AFTER ROTATION (ELASTICSEARCH)

ElasticSearch								
Rotated Component Matrix								
	Component							
	1	2	3	4	5	6	7	8
GodClass	,728							
FeatureEnvy	,698							
IntensiveCoupling	,481							
BlobClass		,716						
TraditionBreaker		,641						
SchizoClass		,411						
InternalDuplication			,763					
SiblingDuplication			,685					
ExternalDuplication				,843				
BlobOperation				,486				
MessageChains					,873			
RefusedParentBequest						,791		
DataClumps						,547		
ShotgunSurgery							,938	
DataClass								,852

TABLE IX
COUPLED FACTOR LOADINGS AFTER ROTATION (EBEHANDLING)

Ebehandling													
Rotated Component Matrix													
	Component												
	1	2	3	4	5	6	7	8	9	10	11	12	13
in_FeatureEnvy	,741												
in_GodClass	,644												
in_DataClass	,620												
in_MessageChains	,535	,445											
DataClass	,400												
in_ExternalDuplication		,904											
in_SiblingDuplication		,876											
BlobOperation			,929										
out_BlobOperation			,914										
in_BlobOperation			,501										
out_GodClass				,856									
out_MessageChains				,830									
out_FeatureEnvy				,529									
InternalDuplication					,912								
ExternalDuplication					,888								
out_InternalDuplication						,864							
out_ExternalDuplication						,856							
SiblingDuplication							,727						
out_SiblingDuplication							,650						
MessageChains							,511						
DataClumps								,798					
out_DataClumps								,686					
in_DataClumps								,460					
out_DataClass									,736				
FeatureEnvy									,665				
out_IntensiveCoupling										,779			
in_IntensiveCoupling										,457			
IntensiveCoupling											,732		
in_InternalDuplication											,429		-,425
GodClass													
out_SchizoClass												,903	
in_SchizoClass													,820

TABLE X
COUPLED FACTOR LOADINGS AFTER ROTATION (MAHOUT)

Mahout
Rotated Component Matrix

	Component												
	1	2	3	4	5	6	7	8	9	10	11	12	13
in_RefusedParentBequest	,700												
in_FeatureEnvy	,678												
in_SiblingDuplication	,569						,422						
out_FeatureEnvy	,492											,413	
in_DataClumps	,453	,442											
in_InternalDuplication		,796											
in_BlobOperation		,682											
in_GodClass		,478			,456								
in_ExternalDuplication		,468											
out_BlobOperation			,938										
out_InternalDuplication			,935										
MessageChains				,898									
out_MessageChains				,894									
in_MessageChains					,713								
in_SchizoClass					,633								
in_DataClass					,432								
out_DataClumps						,717							
out_RefusedParentBequest						,612							
DataClumps						,600							
out_SiblingDuplication							,710						
SiblingDuplication							,690						
out_GodClass								,777					
RefusedParentBequest								,597					
GodClass									,796				
ExternalDuplication													
out_DataClass										,773			
FeatureEnvy									,454	,492			
BlobOperation											,783		
InternalDuplication									,401		,683		
SchizoClass												,742	
DataClass													,745
out_ExternalDuplication													,612

TABLE XI
COUPLED FACTOR LOADINGS AFTER ROTATION (ELASTICSEARCH)

ElasticSearch
Rotated Component Matrix

	Component																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
in_FeatureEnvy	,703																
in_IntensiveCoupling	,665																
in_BlobOperation	,663																
in_SchizoClass	,600																
in_GodClass	,573																
in_DataClumps	,424																
out_IntensiveCoupling		,874															
out_BlobOperation		,869															
out_ShotgunSurgery		,518															
in_SiblingDuplication			,676														
in_MessageChains			,658														
in_InternalDuplication			,512														
in_DataClass			,486														
in_BlobClass				,764													
in_TraditionBreaker				,711													
in_ShotgunSurgery																	
out_DataClumps					,685												
DataClumps					,644												
out_GodClass					,611												
out_TraditionBreaker						,689											
out_InternalDuplication						,618											
out_BlobClass						,528											
out_SchizoClass						,439											
GodClass							,713										
FeatureEnvy							,603										
IntensiveCoupling							,541										
out_DataClass								,661									
out_FeatureEnvy								,499					,454				
BlobClass									,735								
TraditionBreaker									,588								
BlobOperation									,478								
InternalDuplication										,716							
SiblingDuplication										,687							
out_ExternalDuplication											,697						
in_ExternalDuplication											,533						
ExternalDuplication											,480						
out_RefusedParentBequest												,808					
in_RefusedParentBequest												,644					
out_MessageChains													,652				
MessageChains														,683			
out_SiblingDuplication														,495			
SchizoClass															,709		
DataClass																	
RefusedParentBequest																,687	
ShotgunSurgery																	,889