Investigating the Impact of Code Smells Debt on Quality Code Evaluation

Francesca Arcelli Fontana
Department of Computer Science
University of Milano Bicocca
Milano, Italy
arcelli@disco.unimib.it

Vincenzo Ferme
Department of Computer Science
University of Milano Bicocca
Milano, Italy
info@vincenzoferme.it

Stefano Spinelli
Blue Reply S.r.l.
Milano, Italy
st.spinelli@reply.it

***Abstract*—Different forms of technical debt exist that have to be carefully managed. In this paper we focus our attention on design debt, represented by code smells. We consider three smells that we detect in open source systems of different domains. Our principal aim is to give advice on which design debt has to be paid first, according to the three smells we have analyzed. Moreover, we discuss if the detection of these smells could be tailored to the specific application domain of a system.**

Keywords-design debt; code smell refactoring; software quality metrics.

I. INTRODUCTION

If we want to analyze the impact of design debt on the quality of a system, we can perform different tasks, as those described in the paper of Brown et al [5]. The effective managing of debt is perceived to be critical in order to achieve and maintain software quality. Code smells as defined by Fowler [8], reflect code that is decaying and likely to be the cause of frequent and future maintenance activities. For this reason code smells can represent an important source of design debt that have to be managed in some way. One way to eliminate code smell design debt is through refactoring, but refactoring could sometimes be too expensive, and so it is important to focus on the refactoring that could improve a system significantly.

In fact, code smells are not necessarily symptoms of problems and not necessarily have to be removed. For example, a Large Class smell not always has to be refactored, in some cases it could represent the best solution, as for example for a parser implementation.

To decide if smells represent design debt and if they have to be removed or not, we can investigate the correlations among smells and changes or fault proneness, as done in several works in the literature (see Section II-Related Works) or we can analyze the impact the smells have on the quality of a system, in particular on the values of some software quality metrics. In a previous paper [2] we addressed this problem of the impact of refactoring of code smells; here we extend the previous results considering more metrics and more systems of different domains. Hence in our study, starting from the assumption that code smells can represent important sources of design debts, we investigate the relationship between code smells and software quality metrics, and thus indirectly with technical debt.

In this paper we attempt to answer the following questions:

1) What is the impact of removing a particular kind of smell on different software quality metrics? And hence which smells are more critical from the design debt point of view, and which should be removed first? In other words, is there a smell which represents an indicator of design debt more than other smells?

2) Is it possible that a smell detected in a system could not be considered a smell in a system of another domain? Are there some kinds of dependencies of the smell debts to the domain?

According to the first questions, we have considered three smells, which are very common and probably critical smells: Data Class, God Class (in some cases also called Large Class) and Duplicate Code smells. We have detected these smells on different open source systems of different domains, considering three systems for each domain, among the Diagram generator/data visualization, Software Development, Application Software and Client-Server Software domains. We have removed the three smells in the systems by applying the right refactoring steps, and we have computed the value of twelve metrics for software quality evaluation, before and after the refactoring, considering metrics to evaluate cohesion, complexity and dependency. Then we analyzed the impact of refactoring on these metrics, with the aim to evaluate the most dangerous smell and hence the smell which represents the worst technical debt. We could face situations of significant improvement of several metrics, or improvements of some values and the deteriorations of others or no effect at all.

According to the second question, as we observed before for the Large Class smell in the case of a parser, it could be difficult to decide when we have to consider a method long, and the same problem arises for other smells. The size of the system where we detect the smell, the domain of the system, the expertise and personal background knowledge of the software engineers who have to maintain the system have an important role in establishing if a smell really represents a design debt. Usually the tools for code smell detection exploit rules based on a combination of a set of metrics. In some cases, we can simply take into account one of the above features or situations, only by changing the thresholds of the metrics, in other cases other constraints have to be

added to the detection rules of the smell, especially for those pertaining to semantic features. We discuss in the paper, in particular for the God and Data Class, some cases where these two smells are domain-dependent and they should not be considered as indicators of design debts. Our considerations took place, when we checked the precision of the results of smell detection; we found several false positive smells, some of these were domain-dependent smells.

We decided to exploit open source tools both for automatic code smell detection, for the application of automatic refactoring and for metrics computation.

The paper is organized in the following Sections: Section II briefly introduces some related works; Section III describes the smells on which we focus on; Section IV describes the set of metrics we have considered in our experimentation and the systems on which we performed the analysis; Section V reports the results on the detection of the smells, the variation of the metrics values respect to the refactoring of the smells; then we discuss if some domain specific smells exist. Finally, we discuss some threats to validity, we provide our conclusions on code smell debts and we outline some future developments.

II. Related Work

Zazworka et al. [21] investigated the impact of design debt on software quality considering the God Class smell. They outline that God Class smells are more change and defect prone and hence difficult to maintain. In our paper we consider the God Class, Data Class and Duplicate Code and we perform other analyses respect to design debt managing.

Olbrich et al. investigate if code smells are all harmful [18]. In particular, they study God Classes and Brain Classes in the evolution of three open source systems.

Tahvildari et al [19] propose a framework in which a catalogue of object-oriented metrics can be used as an indicator for the transformations to be applied to improve the quality of a legacy system. They use metrics to detect potential design flaws and to suggest transformations to correct them. They consider architectural, structural and behavioral flaws. Here we consider code smells, and we use metrics to evaluate the positive or negative impact of refactoring on the quality of a system and to identify the priorities of the smells to be removed.

Zhang et al [22] investigate the relationship between six code smells (Duplicated Code, Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man) and software faults and discuss how their results can be used by software developers to prioritize refactoring. In particular, they suggest that source code containing Duplicated Code is likely to be associated with more faults. As a consequence, Duplicated Code should be prioritized for refactoring. Respect to our work, we analyze duplicate code and other two smells, considering not the relations with faults, but the impact of the smell refactoring on different quality metric values.

Finally, Arcelli et al [2] propose a first analysis on the impact of refactoring applied to remove smells on a small set of software metrics.

This paper extends the previous one and is different from the other papers cited in this section and from other in the literature in this area. To our knowledge the other works have not considered the impact of refactoring on many software quality metrics; they have also not considered systems of different domains, and hence the role of the domains. Except for one work by Guo et al. [10] where they take into consideration domain-specific characteristics in an iterative empirical study on one software project and exploit their own tool for code smell detection.

III. Code Smells

Smells have been defined by Fowler [8], others have been identified in the literature later [16] and new ones can be discovered. Different tools for code smell detection have been developed that exploit different detection techniques. The detection techniques are usually based on the computation of a particular set of combined metrics [17], standard object-oriented metrics or metrics defined ad hoc for the smell detection purpose.

We tried different tools for code smell detection [3], and we decided to use in particular iPlasma [14] because through this tool we are able to clearly understand, at least for 8 smells, the detection techniques that have been exploited and the metrics thresholds [17]. For the other smells the user is able to define his own detection rule and exploit the metrics computation facility of iPlasma. The evolution and extension of iPlasma has led to the development of the commercial tool InFusion [12].

We decided to focus our attention on the God Class, Data Class and Duplicate Code. The choice of these smells is due to the fact that they are very common, and we got direct proof of this through an experimentation we did on the detection of 14 smells on 68 systems of different domains of the Qualitas Corpus [20].

These smells have been defined in the following way:

Data Class [8]: classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes.

Duplicated code [8] is the most pervasive and pungent smell in software. It tends to be either explicit or subtle. Explicit duplication exists in identical code, while subtle duplication exists in structures or processing steps that are outwardly different, yet essentially the same.

A God Class [17] performs too much work on its own, delegating only minor details to a set of trivial classes and using the data from other classes. This has a negative impact on the reusability and the understandability of this part of the system. This design problem is comparable to Fowler's Large Class bad smell [8].

IV. Metrics Evaluation on Different Systems

In this Section we describe the metrics that we have considered for our experimentation and the systems we have decided to analyze.

A. The Metrics and the Tools

We performed our analysis by considering several metrics related in particular to cohesion, coupling and complexity. We have considered different metrics which describe the features of OO software systems as described in [6, 4, 11] and other metrics introduced in [17], always with the aim to characterize OO systems.

The metrics are outlined in Table I, where we report the name of the metric, the granularity level and the tool we used to compute each metric. We decided to exploit the following tools for metrics computation: iPlasma [14], Google CodePro Analytix [9] and EclipseMetrics [7].

The choice of these tools is due to the fact that it is possible to discover and clearly understand how each metric has been computed and how each metric has been defined.

TABLE I. METRICS AND TOOLS

Metrics	Granularity	Tools
Abstracteness (Abstr)	System	Google CodePro Analytix
Average Line of Code per Method (ALCM)	System	
Distance from Main Sequence (DMS)	System	
CYCLO – McCabe's Cyclomatic Number	Operation	iPlasma
WMC – Weighted Method Count	Class	
AMW – Average Method Weight	Class	
CC – Changing Classes	Method	
FANOUT – Number of Called Classes	System	
ATFD – Access to Foreign Data	Class, Method	
LAA – Locality of Attribute Accesses	Method	
TCC – Tight Class Cohesion	Class	
LCOM – Lack of Cohesion in Method	Method	Eclipse Metrics

B. Systems Analyzed

We decided to consider systems of different domains, as those cited in the Introduction. In particular, in each domain we have analyzed three Java systems, see Table II. These systems have been identified in the Qualitas Corpus of Tempero et al [20] and we have considered in each domain one Small-Medium size system and two Medium size systems, where for Small-Medium we mean 15.000-39.999 LOC and for Medium 40.000-100.000 LOC.

We have decided to perform our analysis by considering for each of the above systems only classes that have been written for this system, e.g. not classes from external libraries (often included in the source code of the downloaded systems), as indicated also by Tempero [20].

V. CODE SMELL REMOVING THROUGH REFACTORING

This section addresses the main contributions of our paper.

A. God Class, Data Class and Duplicate Code smells Detection

We have detected the smells God Class, Data Class and Duplicate Code on all the systems of Table II and we have reported these results in Table III. We used iPlasma for the detection of God and Data Class and Google CodePro Analytics for the detection of Duplicate Code.

The detection of the God Class with iPlasma is done through the computation of a combination of the WMC and TCC metrics and a new one defined in [17] called ATFD (Access to Foreign Data); the detection of the Data Class is always done in the same way but considering other metrics: the WMC, NOPA (Number of Public Attributes), NOAM (Number of Accessor Methods), and WOC (Weight Of Class) metrics. The specific detection rules which are used in iPlasma can be found in [17]. While the detection of the Duplicate Code with CodePro Analytics is done through the computation of the number of lines of code which are exactly equal or just look similar. In the tools settings we have selected these two parameters: the search for complete code fragments (with balanced braces and control structures) and the search for more matches by taking the *much more time option* of the algorithm execution.

From Table III we can observe that we usually have more occurrences of Data Class respect to God Class smell.

TABLE II. SYSTEM ANALYZED

Systems	Applicative Domain	Number of Classes/LOC
Columba 1.0	Application Software	1303/71680
Drawswf 1.2.9	Application Software	302/27008
Galleon 2.3.0	Application Software	556/52653
C_jdbc 2.0.2	Client-Server Software	778/81306
Heritrix 1.8.0	Client-Server Software	649/39272
Struts 2.2.1	Client-Server Software	1608/74670
Ganttproject 2.0.9	Diagram generator/data visualization	801/47051
Jhotdraw 7.5.1	Diagram generator/data visualization	749/75958
Velocity 1.6.4	Diagram generator/data visualization	429/26854
Antlr 3.2	Software Development	330/25243
Drjava 20100913-r5387	Software Development	920/62380
Pmd 4.2.5	Software Development	885/60875

TABLE III. CODE SMELLS OCCURRENCES

Systems	God Class	Data Class	Duplicate Code (LOC)
Columba	14	42	4209
Drawswf	5	35	1376
Galleon	30	41	11556
C_jdbc	30	47	6972
Heritrix	33	18	1529
Struts	36	176	6192
Ganttproject	22	56	1064
Jhotdraw	17	14	9171
Velocity	3	18	1550
Antlr	27	28	3243
Drjava	22	25	5240
Pmd	17	26	2924

B. Metrics Evaluation after Refactoring

For each system and for each smell we applied the refactoring steps of Table IV, and we used the tool IntelliJ IDEA [13] to apply the refactorings.

TABLE IV. REFACTORING APPLICATION

Order	God Class	Data Class	Duplicate Code
1	Extract Class	Encapsulate Field	Extract Method
2	Extract Subclass	Remove Setting Method	-
3	-	Hide Method	-

After applying each refactoring, we checked by running iPlasma again if the refactoring really removed each smell. Moreover, we checked if the refactoring of the God and Data Class introduced a new smell. Only in one case, by removing the smell God Class, a new one, precisely the Brain Class smell [17] was introduced.

In Table V and Table VI we report the results respect to the impact of refactoring of the Data Class and God Class respectively, on the values of the metrics of Table I. We outline the percentage of improvement or deterioration of the value of each metric, through a "+" or "-" sign respectively. In Table VII we report the same results respect to the Duplicate Code refactoring. In this case we have considered only a subset of the metrics of Table I, in particular we have considered two metrics to measure cohesion, two for complexity and two for dependency and we have considered only the systems in the Application Software domain. The metrics of Table I refer to different granularity levels. Hence, the metrics in Table V, VI and VII, which are not at system level granularity, but at method and class level, have been aggregated respect to the mean values.

We have to outline that for the results described in Table V, VI and VII we have checked the results on smell detection of Table III, then we removed the domain specific smells (see Section V C), and we applied the refactoring steps only to the real smells.

We now provide some comments on the results we achieved respect to the refactoring of each smell.

Data Class. Respect to a specific system, as for example the Drawswf system, the WMC, TCC, LCOM metrics improve, while we have a small decrement of the values for LAA, ALCM and CYCLO metrics and a major decrement for the AMW metric. The AMW metric, through the first step of refactoring, improves, because the refactoring Encapsulate Field, has often led to the introduction of methods with low complexity. Then through the second step of refactoring, the AMW metric gets worse, because the refactoring Remove Setting Method has generally led to a reduction of the number of methods. Through the refactoring of the smell, there is an improvement of cohesion of the system, with a small increment in the complexity of the system.

In Heritrix and GanttProject systems we have a similar behavior as for the Drawswf system, in fact, the metrics WMC, TCC and LCOM improve, while instead the metrics LAA, ALCM, CYCLO and AMW get worse. While, if we consider all the systems, we can observe major improvements on the LCOM (6.1 for Jhotdraw), ATFD (5.0% for Struts) and TCC metrics (9.6% for the Struts system), independently from the domain of the systems. This is due to the refactoring technique applied to remove the smell Data Class, which increases the cohesion of the system. We have a small improvement also for the WMC metric, which is used for the detection of the smell. In particular, the refactoring technique, Encapsulate Field, has often led to the introduction of methods of law complexity, which contributes to the improvement of the value of the WMC metric. The values of the LAA, AMW, CYCLO and ALCM metrics, do not improve for the systems in all the domains. In particular, we have a decrement for the ALCM metric, due to the Encapsulate Field refactoring technique, which involved the introduction of new methods with few lines of code.

God Class. Respect to a specific system, as for example the Jhotdraw system, we can observe a negative variation for the TCC metric and a positive variation for the FANOUT, ATFD, LCOM metrics. In general, also for the God Class, we can observe the major improvement for the LCOM (5.8% for Jhotdraw), TCC (9.6% for Struts) and ATFD (7.2% for Struts) metrics, independently from the application domain of the analyzed systems. Also in this case the improvement is due to the applied refactoring technique, which increases the cohesion.

In general, the CYCLO metric does not improve in all the domains, except for the Software Development domain, where the values remain stable. The reason is due to the refactoring techniques applied, which change the control flow graph of the refactored system. Also the AMW metric

does not improve, independently from the domain, because the refactoring techniques involve the introduction of new classes and the movement of some methods and attributes of the refactored classes in these new classes.

When we apply the refactoring steps for the God Class, it would be useful to immediately know the impact on some metrics values. This could be important to exactly identify the right methods and attributes to be moved in other classes. The wrong choice could lead to the deterioration of other metrics values, as those related to coupling. It would be interesting to have the possibility to have this check directly available in a refactoring tool.

Both for God and Data class refactoring, the values of the Abstr and DMS metrics remain the same for almost all the systems. For Data Class, also FANOUT and CC metrics values do not change. This is true because the refactoring techniques of Data Class do not move operations in other classes, then the FANOUT value remains the same, in contrast to the refactoring techniques applied to remove the God Class. Moreover, the techniques used do not introduce any abstract type (hence the Abstractness metric remains stable), and do not have any impact on the metrics used to compute Instability (Afferent and Efferent Coupling Coupling) and consequently also the metric DMS is not affected. The stability of the metric CC, according to the removal of the Data Class, is due to the absence of movements of operations during refactoring.

We can easily see from Table V and Table VI that through the refactoring of the God Class smell, we obtain a major improvement, respect to the Data Class refactoring, on the metric values on the systems of all the domains. However, for the Data Class we have less negative variations, due to the structural differences of the two smells, but the improvements are not relevant as those for the God Class. We report in Fig. 1 and Fig. 2 the sum of all improvements and the sum of all the deteriorations of all the metric values for the refactoring of Data Class (Fig. 1) and for the God Class (Fig. 2). We have considered the systems in the Client-Server software domain because these systems show the major improvements. As we can easily observe from the two figures, for the Struts system we obtain the major improvements.

Duplicate Code. In all the systems, the two metrics that measure the dependence (CC and FANOUT) worsen because often to remove the duplicate code between two different classes we have to create a method in a class and invoke it from the other one. The complexity has a slight improvement due to these cases, in which portions of code with a lot of duplicated logic are reduced to a single method with the same logic. LCOM values decrease in all systems since respect to the initial state of the system, there are more methods that access the same attributes of the refactored class. The TCC metric improves in all the systems, indicating an improvement of the overall cohesion.

C. Domain Specific Code Smells

In this section we try to investigate if a smell could be related to the domain of a system; or better, if it would be possible that a smell is really a smell for a system in a particular domain and not in another one. We investigate if we can provide some kind of domain-based smell filter or if we can find some relationships between smells and frameworks or libraries. If we are able to identify some kind of filter, this could be used to improve the code smell detection techniques used by the tools to remove from the detection domain-specific code smells, which do not represent specific design debts. In this way, we can focus only on the smells which really represent problems.

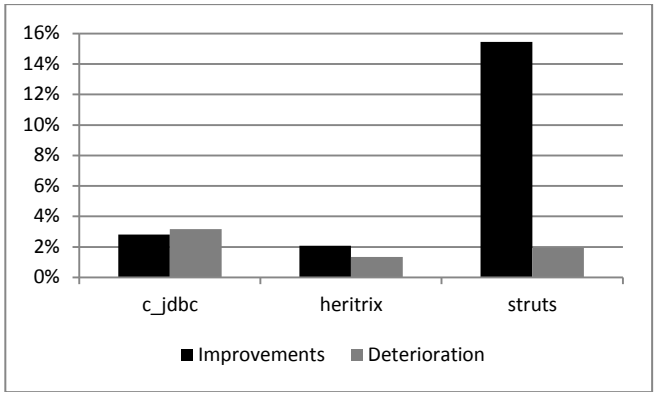

Figure 1. Result on Data Class

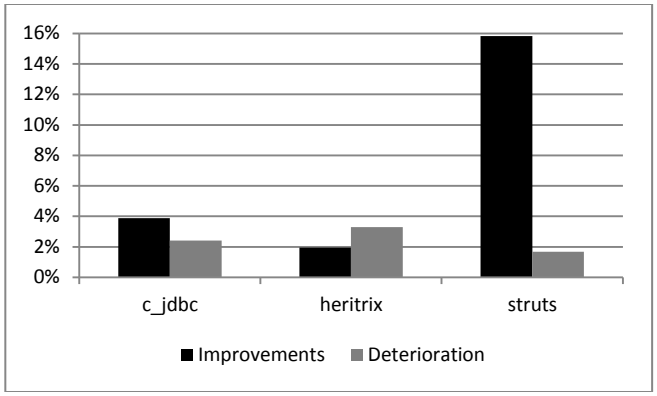

Figure 2. Result on God Class

If we consider for example parser implementations, we often detect God Class smells, also if in this case the smell does not represent a problem or a debt because it could represent the best solution. The systems which we are more interested in this aspect are those in the Software Development and in the Client-Server Software domains.

While if we consider Awt and Swing libraries, these often provide interfaces for the GUI, which are implemented through God Classes. The classes obtained through the tools for the GUI generation have many public methods, and for this reason they can be easily identified as God Classes. In this case it is important to follow the indication of Fowler [8] "especially if you are using older Abstract Windows Toolkit (AWT) components, you might follow this by removing the GUI class and replacing it with Swing-components."

Regarding the Data Class smells, often some classes, such as beans, are identified as Data Class because they have methods that are never used by other classes, but are used at run-time to populate the beans. Very often systems that use

beans also use frameworks which introduce tags that can be used to identify these classes and exclude them from the set of classes detected as Data Class. The systems that are most interested in this aspect are those in the Client-Server Software domain. Some classes often identified as Data Class have the following problems: 1) classes with many getter and setter methods, which are not used and are usually generated by developing tools; 2) the visibility of methods and attributes not correctly managed.

Moreover, another problem when we consider parsers identified as God Classes regards the data structures which they use. These structures are often identified as Data Class smells because the operations on these classes are in the parser class. In these cases, removing the Data Class could be very expensive and not the right choice. In other cases, the classes for the logging or debugging of an application are identified as Data Classes. Also in this case, the refactoring to remove these classes could not represent an improvement.

As we have seen, there are many cases of smells that are not really smells. The detection rules of smell are valid but do not always lead to results that are useful for refactoring. By using the current rules for smell detection provided by the tools and then applying filters based on the information we just provided, it allows us to obtain more reliable results for the evaluation of the smells to be refactored according to the features of the domain.

VI. Threats to Validity

For what concerns the threats to validity, we can assert that we have analyzed systems of different sizes and in four different domains. All these systems are open source systems and have been selected in the Qualitas Corpus [20]. The tools we have used during our experimentation are open source tools. Hence, all the aspects related to the replication of data are preserved.

For what concerns the validity of the tools for code smell detection: we checked all the results on the detection provided by iPlasma and Google CodePro for what concerns false positives. We have not checked if there are other smells not detected by the tool (false negatives). This check is difficult because it involves the problem related to the detection rule and the threshold setting values of the metrics. However, in any case, our aim was not to compare or validate the results of code detection tools.

For the validity of the automated refactoring tools, we checked if the refactoring transformation to remove a smell behaved correctly; moreover we checked if the refactoring of the God and Data Class has introduced other smells.

Finally, for the validity of the metrics computation tools, we supposed that the tool Google CodePro, Eclipse Metrics and iPlasma which we used, provided safe computation, but we have not compared the values obtained with those obtained with other tools.

Obviously, the analysis we have made has to be extended, considering the impact of refactoring of other smells and on other systems in the four domains. Here we have considered only three smells, but they represent very common smells.

TABLE V. Results on Data Class

Systems	Metrics											
	ALCM	**CYCLO**	**WMC**	**AMW**	**Abstr.**	**DMS**	**CC**	**FANOUT**	**ATFD**	**LAA**	**TCC**	**LCOM**
Columba	= 0%	= 0%	+ 0,85%	- 0,36%	= 0%	= 0%	= 0%	= 0%	= 0%	- 0,07%	+ 1,04%	= 0%
Drawswf	- 1,16%	- 1,10%	+ 1,17%	- 2,65%	= 0%	= 0%	= 0%	= 0%	= 0%	- 0,02%	+ 3,43%	+ 5,59%
Galleon	- 0,45%	- 0,41%	+ 0,27%	- 0,05%	= 0%	= 0%	= 0%	= 0%	= 0%	- 0,08%	- 0,01%	+ 0,50%
C_jdbc	- 1,47%	- 0,93%	+ 0,87%	- 0,35%	= 0%	= 0%	= 0%	= 0%	- 0,34%	- 0,09%	+ 0,76%	+ 1,19%
Heritrix	- 0,42%	- 0,51%	+ 0,26%	- 0,38%	= 0%	= 0%	= 0%	= 0%	= 0%	- 0,04%	+ 0,48%	+ 1,33%
Struts	- 0,25%	= 0%	- 0,01%	- 0,04%	= 0%	= 0%	- 0,67%	- 0,83%	+ 5,00%	- 0,13%	+ 9,68%	+ 0,77%
Ganttproject	- 0,40%	- 0,65%	+ 0,41%	- 0,09%	= 0%	= 0%	= 0%	= 0%	= 0%	- 0,04%	+ 0,63%	+ 0,87%
Jhotdraw	- 0,41%	- 0,52%	+ 0,30%	- 0,05%	= 0%	= 0%	+ 0,08%	= 0%	= 0%	- 0,01%	+ 0,44%	+ 6,10%
Velocity	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%
Antlr	- 0,66%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	+ 4,80%
Drjava	- 0,12%	= 0%	+ 0,18%	- 0,10%	= 0%	= 0%	= 0%	= 0%	+ 4,44%	- 0,02%	+ 0,34%	+ 0,96%
Pmd	- 0,84%	- 0,88%	+ 0,62%	- 0,33%	= 0%	= 0%	= 0%	= 0%	= 0%	- 0,10%	- 0,14%	+ 1,53%

TABLE VI. RESULTS ON GOD CLASS

Systems	Metrics											
	ALCM	CYCLO	WMC	AMW	Abstr.	DMS	CC	FANOUT	ATFD	LAA	TCC	LCOM
Columba	+ 0,60%	= 0%	+ 0,85%	- 0,36%	= 0%	= 0%	= 0%	= 0%	= 0%	- 0,07%	+ 1,04%	+ 0,65%
Drawswf	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%
Galleon	- 0,22%	- 0,41%	+ 0,49%	- 0,15%	= 0%	= 0%	+ 0,18%	+ 0,97%	+ 1,28%	- 0,19%	- 3,03%	+ 0,53%
C_jdbc	- 0,81%	- 0,46%	+ 1,26%	- 0,70%	= 0%	= 0%	- 0,53%	- 0,21%	- 0,45%	- 0,08%	+ 1,45%	+ 1,18%
Heritrix	= 0%	- 0,51%	+ 0,62%	- 2,07%	= 0%	= 0%	- 0,14%	- 0,17%	- 0,36%	- 0,06%	+ 0,04%	+ 1,29%
Struts	= 0%	= 0%	- 0,01%	- 0,04%	= 0%	= 0%	- 0,67%	- 0,83%	+ 5,00%	- 0,13%	+ 9,68%	+ 1,15%
Ganttproject	+ 0,40%	= 0%	+ 0,45%	- 0,23%	= 0%	= 0%	- 0,46%	- 0,58%	- 1,33%	= 0%	+ 1,07%	+ 0,70%
Jhotdraw	= 0%	- 0,52%	+ 0,41%	- 0,05%	= 0%	= 0%	- 0,02%	+ 2,39%	+ 7,28%	+ 0,26%	- 10,62%	+ 5,86%
Velocity	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%	= 0%
Antlr	= 0%	+ 1,06%	- 6,47%	- 3,72%	= 0%	= 0%	- 4,25%	- 5,35%	- 4,11%	- 3,05%	+ 1,53%	+ 4,68%
Drjava	+ 0,71%	= 0%	+ 0,23%	- 0,37%	- 1,41%	= 0%	- 0,59%	- 1,41%	- 6,49%	- 0,01%	+ 0,61%	+ 0,88%
Pmd	- 0,36%	- 0,88%	+ 0,74%	- 0,94%	= 0%	= 0%	- 1,10%	- 1,16%	- 1,64%	- 0,09%	- 0,08%	+ 1,51%

TABLE VII. RESULTS ON DUPLICATE CODE

Systems	Metrics					
	WMC	CYCLO	LCOM	TCC	CC	FAN
Columba	+ 0,5%	= 0%	- 0,9%	+ 3,3%	- 0,6%	- 3,4%
Drawswf	+ 0,6%	+ 1,5%	- 1,6%	+ 1,5%	- 1,3%	- 0,3%
Galleon	+ 1,2%	+ 1,2%	- 2,1%	+ 2,2%	- 0,3%	- 2,7%

VII. CONCLUSIONS ON CODE SMELL DESIGN DEBT

In this paper we have tried to investigate aspects related to the following questions:

1) Can we identify smells that represent more critical debts respect to others?

The Duplicate Code smell is certainly one of the most dangerous smells which has to be removed first or better before other smells, but as outlined also in [15], we can find several situations where code duplication seems to be a reasonable or even beneficial design option. The results that we have obtained in this paper outline that the refactoring of the duplicate code led to an improvement of the cohesion and complexity, and hence of the maintainance of the system. Among the smells we have considered, this is the only one, whose refactoring led to the improvement of the CYCLO metric. The Extract Method refactoring technique has a negative impact on system dependency. We have only used this technique because most of the duplicate code has been identified in the same classes or in classes that already had a kind of dependency. The global dependency worsens due to a few cases in which the extract method refactoring creates new methods that introduce dependency between classes previously independent. In order to maintain a good level of global dependency we could also use in a future work the Extract Class refactoring proposed by Fowler.

The refactoring of the other two smells in general has a positive impact in particular on the following metrics WMC, TCC, LCOM metrics and a negative impact on ALCM (in particular for Data Class), CYCLO, AMW, LAA metrics. They have to be considered as good candidates for the refactoring and the right managing of technical debt. It is difficult to indicate the order in which the three analyzed smells should be refactored. In order to achieve such prioritization, our analysis must be extended to a larger number of smells. We can suggest which smell should be removed first, depending on the OO software system features that we need to improve first and on the refactoring effort. If the priority is to improve the cohesion of the system, then both the Data Class and God Class smells have to be removed because they have a great improvement for LCOM and TCC metrics. However, through the God Class refactoring we have the greatest deterioration of CC, FANOUT and ATFD metrics and more effort for the refactoring.

2) Can we identify smells which are domain-dependent, and should they be not considered code smell debts in the specific domain?

Yes, as we have discussed in Section V-C, both the God Class and Data Class are smells that could be domain-dependent, and we reported several examples of them. In particular, this is due to the libraries and frameworks used that are related to the features of the domain. We have

detected the smells through iPlasma, and then we manually checked all the smells. During this step, we realized the importance of taking into account the domain or other features, which the tools are not able to take into consideration. We found many false positives, as in the case of data beans for the Data Class or classes that use Swing and Awt library or implement parser for the God Class.

We hope that the investigation described in this paper is useful to provide some hints on the first smells to be removed in a system to better manage code smell debt and to improve code and design quality.

VIII. Future Developments

A future area of investigation is related to the identification of the most frequent smells in the systems and also the domain characterized by the largest number of smells. We have already started this analysis on 68 systems of different domains and with the detection of 12 smells.

Moreover, it would be interesting to study what is the right order of smells to be refactored to obtain the best improvements on the software quality metrics.

Among the future developments, we obviously have to consider other smells, other systems and other refactorings; we also aim to study the effects of the refactorings on the average time to fix a defect and on the reduction in bugs.

We are currently working on a code smell detector, developed as a module integrated into our project Marple [1] (Metrics and Architecture Reconstruction PLug-in for Eclipse). We would like to refine the detection rules of our code smell detector and take into account the knowledge on the domain and libraries, with the aim of detecting and then removing only the smells which really represent debts. In Marple we exploit data mining techniques for design pattern detection; it could be interesting to explore if some kind of classification or mining techniques can be exploited to refine and improve smell detection. Moreover, a module of Marple has been developed for metric computations, currently used for software architecture reconstruction and anti-pattern detection. We aim to extend this module for the computation of the metrics cited in this paper and other metrics useful for our investigation. This module is integrated into Marple and could immediately provide data on the variations of the different metric values, respect to the refactoring of one or other smells. These data could be also used to provide correlation analysis among the different metrics.

References

[1] F.Arcelli Fontana and M. Zanoni, "A tool for design pattern detection and software architecture reconstruction," Information Science Journal, Elsevier, vol. 181, April 2011, pp. 1306-1324, doi:10.1016/j.Dec.2011.

[2] F.Arcelli Fontana and S.Spinelli, "Impact of Refactoring on Quality Code Evaluation," Proceedings of the 4th Workshop on Refactoring Tools (WRT 11), ACM, May 2011, pp. 37-40, doi:10.1145/1984732.1984741.

[3] F.Arcelli Fontana et al, "An experience report on using code smells detection tools," Proceedings of International Conference on Software Testing, Verification and Validation Workshops (ICSTW 11), IEEE Press, March 2011, pp. 450-457, doi:10.1109/ICSTW.2011.12

[4] L.C.Briand, J.Wüst, J.W.Daly and D.V.Porter ,"Exploring the Relationship between Design Measures and Software Quality in Object-Oriented System," Journal of Systems and Software, vol. 51, May 2000, pp. 245-273, doi:10.1016/S0164-1212(99)00102-8

[5] N.Brown et al, "Managing technical debt in software-reliant systems," Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER 10), ACM, 2010, pp. 47-52, doi: 10.1145/1882362.1882373

[6] S.R.Chidamber and C.F.Kemerer, "A metric suite for object-oriented design," IEEE Transactions of Software Engineering, vol. 20, June 1994, pp. 476-493, doi:10.1109/32.295895

[7] Eclipse Metrics plugin: http://eclipse-metrics.sourceforge.net

[8] M.Fowler et al, Refactoring: Improving the Design of Existing Code, Addison-Wesley., Inc., Boston, MA, USA, 1999.

[9] Google CodePro Analytix: http://code.google.com/intl/it-IT/javadevtools/codepro/doc/index.html

[10] Y.Guo, C. Seaman, N. Zazworka and F. Shull, "Domain-specific tailoring of code smells: an empirical study," Proceedings of 32nd International Conference on Software Engineering (ICSE 10), ACM/IEEE, May 2010, pp. 167-170, doi:10.1145/1810295.1810321

[11] B.Henderson-Sellers, Object-Oriented Metrics : Measures of Complexity, Prentice Hall, 1996

[12] InFusion: www.intooitus.com/inFusion.html

[13] IntellJ Idea 10:
http://www.jetbrains.com/idea/features/index.html

[14] iPlasma: http://loose.upt.ro/iplasma/index.html

[15] C.Kapser and M.W.Godfrey, "Clones considered harmful" considered harmful, Proccedings of 13th Working Conference on Reverse Engineering (WCRE 06), IEEE Computer Society, 2006, pp. 19-28, doi:10.1109/WCRE.2006.1

[16] J.Kerievsky, Refactoring to Patterns, Addison Wesley, 2005.

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

[18] S.M.Olbrich, D.S.Cruzes and D.I.K.Sjoberg, "Are all code smells harmful? A study of God Classes and BrainClasses in the evolution of three open source systems," Proceedings of International Conference on Software Maintenance (ICSM 10), IEEE, Sept. 2010, pp. 1-10, doi:10.1109/ICSM.2010.5609564

[19] L.Tahvildari and K.Kontogiannis, "A Metric-Based Approach to Enhance Design Quality Through Meta-Pattern Transformations," Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR 03), IEEE Computer Society, March 2003, pp. 183-192, doi:10.1109/CSMR.2003.1192426

[20] E.Tempero et al, "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies," Proceedings of 17th Asia Pacific Software Engineering Conference (APSEC 10), IEEE Press, Dec. 2010, pp. 336-345, doi:10.1109/APSEC.2010.46

[21] N.Zazworka, C.Seaman and F.Shull, "Prioritizing Design Debt Investment Opportunities," Proceedings of the 2nd Workshop on Managing of Technical Debt (MTD 11), ACM, May 2011, pp. 39-42, doi: 10.1145/1985362.1985372

[22] M.Zhang, N.Badoo, P.Wernick and T.Hall, "Prioritizing Refactoring using code bad smells," Proceedings of 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW 11), IEEE, March 2011, pp. 458-464, doi:10.1109/ICSTW.2011.69