Investigating the Evolution of Bad Smells in Object-Oriented Code

Alexander Chatzigeorgiou   Anastasios Manakos

Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
{achat, mai0932}@uom.gr

Abstract— **Software design problems are known and perceived under many different terms such as bad smells, flaws, non-compliance to design principles, violation of heuristics, excessive metric values and antipatterns, signifying the importance of handling them in the construction and maintenance of software. Once a design problem is identified, it can be removed by applying an appropriate refactoring, improving in most cases several aspects of quality such as maintainability, comprehensibility and reusability. This paper, taking advantage of recent advances and tools in the identification of non-trivial bad smells, explores the presence and evolution of such problems by analyzing past versions of code. Several interesting questions can be investigated such as whether the number of problems increases with the passage of software generations, whether problems vanish by time or only by targeted human intervention, whether bad smells occur in the course of evolution of a module or exist right from the beginning and whether refactorings targeting at smell removal are frequent. In contrast to previous studies that investigate the application of refactorings in the history of a software project, we attempt to study the subject from the point of view of the problems themselves distinguishing deliberate maintenance activities from the removal of design problems as a side effect of software evolution. Results are discussed for two open-source systems and three bad smells.**

Keywords: bad smell; refactoring; software repositories; software history; evolution

I. INTRODUCTION

The design of software systems can exhibit several problems which can be either due to inefficient analysis and design during the initial construction of the software or more often, due to software ageing, where software quality degenerates over time [20]. The importance of detecting design problems is evident from the multitude of terms under which they are known. Some researchers view problems as non-compliance with design principles [14], violations of design heuristics [22], excessive metric values, lack of design patterns [8] or even application of anti-patterns [3]. According to Fowler [7], design problems appear as "bad smells" at code or design level and the process of removing them consists in the application of an appropriate refactoring, i.e. an improvement in software structure without any modification of its behavior. Refactorings have been widely acknowledged mainly because of their simplicity which allows the automation of their application. Moreover, despite their simplicity, the cumulative effect of successive refactorings on design quality can be significant.

The design quality of a given system is usually assessed by analyzing the current version at hand. However, organized collections of software repositories offer an additional, rich source of information regarding software quality since they grant access to previous versions of the source code. An entire field of research, namely the Mining of Software Repositories (MSR) [11] has focused on the exploitation of past software related data, to support the maintenance of software systems, improve software design/reuse, and empirically validate novel ideas and techniques.

Historical data regarding source code also reflect architectural decisions by recording the evolution of the design and therefore can be valuable in the assessment of maintainability. Several reliable approaches have been developed in order to detect changes and refactorings that have been applied during the history of software projects. The corresponding tools have enabled empirical studies that assessed the employed refactoring practices. In this paper we present the results of a case study on the presence and evolution of three bad smells regarding design issues, by looking at various past versions of two open-source systems. The tool that has been employed is JDeodorant [10] which allows the identification of three non-trivial bad smells, namely *Long Method*, *Feature Envy* and *State Checking*. In contrast to previous studies that mainly focused on the identification of refactorings, the results emphasize findings and assumptions regarding the problems themselves and the reasons causing their appearance and removal during software evolution. The goal of this study is to shed light on questions such as:

- Does the number of design problems increase over time?

- Will the evolution of a software system remove some of its bad smells or are the problems solved only after targeted maintenance activities?

- Do bad smells exist in a software module right from its initial construction or do they appear during its evolution?

- How frequent are refactoring activities that target bad smells?

- How urgent is it to remove the identified code smells?

IEEE
computer
society

The findings which are being discussed in this paper, at a first-level can be considered as project-related, in the sense that they characterize aspects of the design quality for the particular systems that have been studied. However, they also provide initial evidence regarding the refactoring practices (identification and handling of smells) that have been followed during the history of the examined projects. In this context, the results of the study provide information regarding the culture, skills and attitude towards refactorings of the development team, although further studies are required to validate such generalizations.

The rest of the paper is organized as follows: Section II discusses related work on refactoring identification approaches, empirical studies regarding refactoring practice and tools that allow the detection of code smells. Section III describes briefly the essence of the bad smells that have been investigated and the projects that have been selected. In section IV bad smell categories are defined and the results are presented in visual and tabular form and findings are being discussed. The analysis focuses on the total number of smells, their persistence and evolution as well as the urgency to remove them. Threats to validity are presented in section V. Finally, we conclude in section VI.

II. RELATED WORK

A number of studies have focused on the detection of changes and refactorings that have been applied in past versions of software projects acknowledging that historical data are valuable during maintenance.

Demeyer et al [4] presented a metrics-based approach for refactoring identification. Metric values concerning method size, class size and inheritance are collected for two successive versions of a given system. The refactoring operations that have been applied can be identified with the help of heuristics defined as combinations of change metrics. According to the evaluation on three case studies the approach has a good precision and moreover has the advantage of focusing only on relevant parts of the system.

Dig et al. [5] acknowledged the need to identify refactorings performed during component upgrade, a task that is more challenging than detection of refactorings on products of in-house software development. The proposed algorithm detects possible sequences for seven types of refactorings between two versions of a component. The first stage of the algorithm employs similarity techniques to identify similar fragments of source code entities which are candidates for refactorings. The second stage employs semantic analysis to detect from the candidate pairs the cases where one entity is a likely refactoring of the other. Evaluation on three real-world components showed that the algorithm achieves accuracy over 85%.

A design-level differencing methodology to recognizing applied refactorings has been proposed by Xing and Stroulia [30]. The approach employs UMLDiff, a domain-specific differencing algorithm that detects numerous kinds of elementary structural changes. Applied refactorings are viewed and detected as compositions of elementary changes. Results from case studies on several releases of two open-source projects revealed that all of the documented refactorings were recovered, while many undocumented refactorings were also identified.

Refactoring identification approaches and tools enabled researchers to perform empirical studies in order to investigate whether refactorings are performed regularly and systematically and to explore programmers' and maintainers' habits regarding refactoring practice.

Xing and Stroulia [29] conducted a case study on the structural evolution of Eclipse to investigate what fraction of code modifications are refactorings and which are the most frequent ones. Their findings indicated that about 70% of structural changes may be due to refactorings. This high frequency of refactorings is probably due to the advanced state of Eclipse in terms of design quality but it remains unanswered whether the applied refactorings are targeted at removing specific bad smells.

An extensive study of refactoring application has been presented by Murphy-Hill et al. [18] based on four sets of data, including data from Eclipse IDE users who submitted refactoring commands back to the Eclipse Foundation and data from the repositories of Eclipse and JUnit. Among the various findings of this study it was observed that refactorings are practiced frequently and more importantly, programmers frequently floss refactor, that is, they mix refactoring with other programming activities regularly. It is worth mentioning that according to the study even medium-level refactorings such as *Extract Method* have been applied frequently but it is unknown whether the refactoring efforts targeted identified design issues and especially non-trivial problems, such as the ones discussed in this paper. (According to the classification assumed in [18], medium-level refactorings are those that change the signatures of classes, methods and fields and also significantly change blocks of code). Refactoring identification from version systems of five open-source projects has also been performed in [21] to investigate the relation between refactorings and probability of future software defects. Identification was based on the textual analysis of messages attached to commits, an approach that has been questioned for its accuracy by Murphy-Hill et al. [18].

Recently, a number of researchers investigated the impact of code smells on change-proneness. Olbrich et al. [19] analyzed the historical data of two open-source projects focusing on the *God Class* and *Shotgun Surgery* code smells. An important conclusion of their analysis was that the evolution of a system undergoes different phases in which the number of smells could be increasing or decreasing. As a result, an overall conclusion regarding the question whether the total number of smells increases steadily or not could not be safely reached. With regard to change behavior, it was observed that the classes infected by the examined smells suffer more changes than the non-infected ones.

A similar conclusion was reached in [12], where statistical analysis of 29 code smells in several releases of two open-source projects revealed that classes with smells are more likely to be the subject of changes. In this context, it is claimed that smells might be more valuable to the developers since they

provide recommendations that are easier to understand than metric values.

A similar study on the evolution of problems but in a different domain was reported by Di Penta et al. [6]. The presented empirical study aimed at analyzing the evolution of source code vulnerabilities, detected by static analysis tools, on three open source network systems. Similar questions such as how long vulnerabilities tend to remain in the system and how vulnerabilities tend to be removed have been investigated. However, according to the statistical results, the vast majority of vulnerabilities, in contrast to bad smells, tend to be removed from the system, implying a different treatment against security issues.

The increased interest in refactorings as a means of improving the design quality is evident from the support that is being offered by state-of-the-art Computer-Aided Software Engineering (CASE) tools. Apart from tools that automate the application of refactorings relieving designers from the burden of refactoring mechanics, recent approaches aim at the development of tools for the identification of design problems and flaws which constitute refactoring opportunities. Without aiming at a thorough survey of the field, noteworthy tools include: ProDeOOS [13] which employs selected metrics to identify suspect classes that might exhibit design problems, such as god and data classes, jCOSMO [28] and its successor CodeNose [23] where identification of a code smell is assumed when all associated smell aspects are found using static analysis, iPlasma [24] which uses a detection strategy based on the composition of various metric rules combined with AND/OR operators in order to express design heuristics, DÉCOR [16], [17] which employs a metrics-based detection approach and allows the specification of smells using a domain-specific language in the form of rules, and Borland Together [2] which also relies on a combination of metrics and predefined threshold values.

III. BAD SMELLS AND PROJECTS

As already mentioned this study employs JDeodorant for the identification of bad smells. The main reason is that the tool offers the possibility to detect non-trivial bad smells whose removal requires a systematic and elaborate refactoring action. In other words, we avoided looking at refactoring opportunities calling for refactorings with simple mechanics, such as *Rename Method* or *Encapsulate Field*, to clearly distinguish cases that correspond to intentional removal of a bad smell. The three bad smells that have been studied are:

A. Long Method

Methods suffering from the *Long Method* bad smell are usually pieces of code with large size, high complexity and low cohesion which consequently require more time and effort for comprehension, debugging, testing and maintenance. (In the context of the *Long Method* smell, cohesion refers to intra-method cohesion expressed for example by slice-based cohesion metrics [15]). An ideal solution to this kind of design problems is given by the *Extract Method* refactoring [7] which simplifies the code by breaking large methods into smaller ones and creates new methods which can be reused.

JDeodorant identifies *Long Method* bad smells and in particular detects refactoring opportunities which a) extract the complete computation of a given variable into a new method [25] and, b) extract the statements affecting the state of a given object into a new method. In the first case a slice that contains all the assignment statements of a given variable within the body of a method is extracted, while in the second case a slice that contains all statements modifying the state of a given object (by method invocations through references pointing to this specific object) is extracted. The identification is performed automatically in the sense that the designer does not have to specify the seed statements for which a slice of code is suggested to be extracted as a new method. Refactoring suggestions are ranked according to the number of duplicated statements (in the original and extracted method) and the number of extracted statements.

B. Feature Envy

Feature Envy is a sign of violating the principle of grouping behavior with related data and occurs when a method is "more interested in a class other than the one it actually is in" [7]. Since *Feature Envy* implies coupling and/or cohesion problems, its presence affects negatively the maintainability of the involved methods and classes. *Feature Envy* problems can be solved either by moving a method to the class that it envies (*Move Method* refactoring) or by moving an attribute to the class that envies it (*Move Field* refactoring).

JDeodorant detects *Feature Envy* bad smells as opportunities for *Move Method* refactoring [26]. Automatic identification is performed employing the notion of distance between an entity (attribute or method) and a class; if the distance of a method to another class is lower than the distance from the class it belongs to, a suggestion is extracted. The distance between a method and a class is defined by the dissimilarity of their entity sets, where the entity set of a method contains all accessed methods and attributes, whereas the entity set of a class contains all of its members [26]. The suggested refactoring opportunities are ranked according to the improvement that they can induce into the design quality, measured by a combined coupling and cohesion metric.

C. State Checking

State Checking (known under the name *Switch Statements* in [7]) manifests itself as conditional statements that select an execution path based on the state of an object. In the usual scenario the associated *switch* or *if/else* statements are scattered in different places of the program. The existence of *State Checking* actually represents a missed opportunity for applying polymorphism or in other words the lack of the *State/Strategy* design pattern. The presence of this smell essentially signifies a violation of the Open-Closed Principle [14] since any future modification in the actions associated with a particular state or the addition of new states will require the modification of existing code increasing the required effort and the possibility of introducing errors.

JDeodorant identifies *State Checking* bad smells as opportunities for introducing polymorphism [27]. The identification is performed by looking for conditional statements that select an execution path either by comparing

the value representing the current state of an object with a set of named constants, or by retrieving the actual subclass type of a reference through Run Time Type Identification (RTTI) mechanisms. Refactoring suggestions are ranked according to the number of occurrences of the *State Checking* smell (which is equivalent to the number of times that the introduced polymorphism will be exploited throughout the system) and the average number of statements that will be moved to the subclasses of the introduced hierarchy.

In the presented case study, results have been obtained for two open-source projects: a) JFlex, which is lexical analyzer generator for Java (analysis has been performed for package JFlex, consisting of 40 classes in the latest version that has been examined) and b) JFreeChart, which is a Java chart library (analysis has been performed for package com.jrefinery.chart consisting of 110 classes in the latest examined version). Bad smells have been identified in 10 versions of JFlex (1.3 – 1.4.3) and 14 versions of JFreeChart (0.5.6 - 0.9.4a). The projects under study had to be written in Java since JDeodorant analyzes Java source code. Moreover, they have been selected because: a) they provide several versions in their repositories and, b) they are mature in the sense that they have a sufficient development time extending for more than 9 years, providing room for refactoring activities. The size characteristics (thousand lines of code and number of classes) of the packages that have been examined in each version of both projects are shown in Table I.

TABLE I. Size Characteristics of the Examined Versions/Projects

Measures	1.3	1.3.1	1.3.2	1.3.3	1.3.4	1.3.5	1.4	1.4.1	1.4.2	1.4.3
JFlex										
kLOC	7.14	7.36	7.48	8.08	8.11	8.15	9.16	9.03	9.62	9.62
NOC	34	34	34	35	35	35	40	40	40	40

Measures	0.5.6	0.6.0	0.7.0	0.7.1	0.7.2	0.7.3	0.7.4	0.8.0	0.8.1	0.9.0	0.9.1	0.9.2	0.9.3	0.9.4a
JFreeChart														
kLOC	5.80	9.00	11.0	10.8	11.4	11.6	11.9	12.1	13.8	19.5	19.7	20.8	25.2	28.8
NOC	47	59	75	66	68	68	70	72	77	99	99	103	106	110

IV. Results And Discussion

A. Total Number of Bad Smells

It is reasonable to expect that since functionality is enhanced in every new version of a software system and since open-source software does not undergo systematic preventive maintenance, the total number of design smells will increase with time. The results, summarized in Figures 1 and 2 confirm this belief, for both systems and all three of the selected bad smells. The number of *Long Method* smells is considerably larger indicating that overly long, complex and non-cohesive methods are more common than the other two symptoms. In almost all cases the number of problems increases as the system evolves, although the rate of increase is lower for *Feature Envy* and *State Checking* smells.
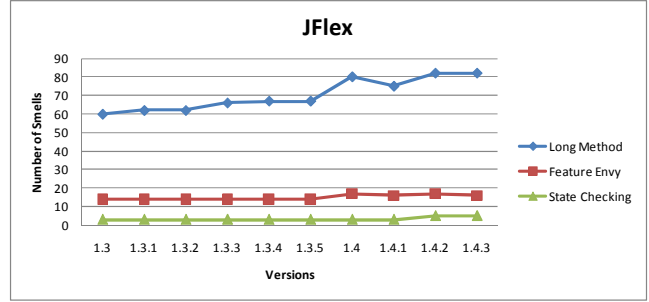

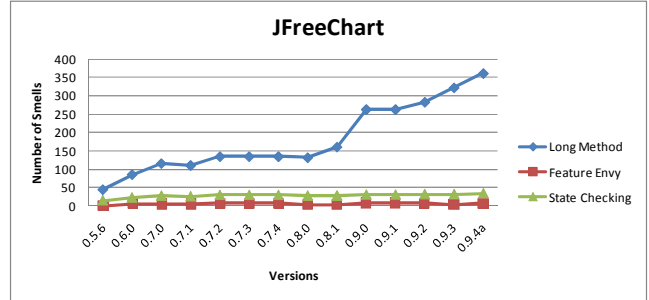Figure 1. Total Number of smells in project JFlex

Figure 2. Total Number of smells in project JFreeChart

B. Persistence of Bad Smells

To obtain an overview of the way design problems evolve over time, we have plotted for project JFlex, in Figure 3, the way in which *Long Method* bad smells spread over successive versions. Each horizontal grey bar corresponds to an identified bad smell and indicates the versions at which the smell was present. The right dashed vertical line corresponds to a hypothetical version following the last one that has been examined, so that each version is represented as an interval up to the next one. From this drawing it becomes apparent that for the overwhelming majority of bad smells (89.8%), once they appear in a certain version, they persist up to the latest version of the project. This fact possibly implies that design problems are lasting and do not vanish unless targeted refactoring activities are performed. As it can be observed, a large portion of the smells (57.7%) are present throughout all of the examined versions. (The total number of distinct smell cases for both projects and all three smells is shown in the last row of Table II).

Regarding the relatively few cases where the existence of a bad smell was terminated in a version, after careful inspection of the source code, the elimination of the problem can be attributed to the following coarse reasons:

- Code rewriting: the code fragment suffering from a bad smell in a previous version has been rewritten, however, with no indication of a refactoring activity. In most cases, rewriting is a behavior-changing activity whereas refactoring is not. A usual case in the systems that we have examined involved complex conditional expressions in which one part contained a variable assignment. Removing the corresponding part of the conditional (for behavior-related reasons) eliminated the *Long Method* bad smell. Thus, we consider these cases as accidental elimination of the smell.

Figure 3. Evolution of *Long Method* bad smells in project JFlex

- Code removal: the entire code fragment suffering from a bad smell in a previous version has been removed from the code base. These cases are also not considered as intentional maintenance targeting at the problem since the elimination of the problem was caused by a change in the provided functionality.

- Class/Method removal: similar to the previous case, but here the entire method or class containing the problem has been removed. Once again, these changes cannot be considered as intentional maintenance activities to remove the smell.

- Intentional Refactoring Activity: These are the cases where the source code of the first problem-free version appears to have undergone a systematic, by-the-book refactoring activity which removed the code smell that was present up to the exactly previous version. For a *Long Method* bad smell, an unambiguous refactoring consists in the extraction of the computation of a variable (or of the statements that affect the state of a common object) as a separate method that is invoked in the original method. For a *Feature Envy* smell, refactoring activity is indicated when the method exhibiting envy to the methods or attributes of another class has been moved to that target class. For a *State Checking* smell the clear sign of a refactoring activity is the introduction of polymorphism to replace the entire suffering conditional expression. However, as it will be shown next, the cases where an unambiguous refactoring activity was identified appear to be

exceptions, since their frequency is very low (on average, 0.77% of all cases).

C. Evolution of Bad Smells

To provide insight into the mechanisms that generate bad smells or cause them to vanish we visualize in more detail the way bad smells appear, sustain and disappear during the course of software versions. We have grouped bad smells into the following categories (the definition of each category becomes clear with its visual representation in Figures 4 and 5):

- A: Smells that exist throughout all examined versions of a project.

- B: Smells that appeared in one of the examined versions (not the first one) and remain up to the latest version. This category can be further decomposed into the following cases with regard to the exact point of "birth" of the smell:

  - B_1: Smells that appeared at a point during the evolution of a project but did not exist when the method in which they reside was introduced. These cases imply that the particular design problem was introduced during evolution or maintenance of the method under study.

  - B_2: Smells that exist right from the beginning of the corresponding method, that is, from the point at which the method in which they reside has been introduced to the system.

- C: Smells which are present from the first examined version but have disappeared in a later version. This category can be further decomposed into the following two cases with regard to the reason that caused the removal of the smell.

  - C_1: Bad smells that have been removed whereas the corresponding method in which they reside remained in the system. Although these cases appear to be successful in terms of improving software quality, after careful examination very few of these cases consisted in an unambiguous refactoring application.

  - C_2: Smells that exist right from the first version that has been analyzed and have been eliminated from the project because the corresponding method has been removed from the system. Obviously, these cases cannot be considered as successful refactoring applications since the method that presented the smell has been completely eliminated.

- D: Bad smells that appeared and disappeared during the course of software versions (not at the first and last version, respectively). This category encompasses four sub-categories with regard to the method containing the problem:

  - D_1: The smell appeared when the corresponding method was introduced. The smell disappeared when the method was removed from the system.

  - D_2: The smell appeared during the evolution of the method (i.e. as a result of its adaptive or corrective

maintenance) and disappeared when the method was removed.

- D_3: The smell appeared when the corresponding method was introduced. The method continued to exist after the removal of the smell.

- D_4: The smell appeared and disappeared during the evolution of the method (i.e. the method existed before the introduction of the smell and after its removal). Cases belonging to D_3 and D_4 categories can potentially be regarded as successful code removal activities.

The results concerning the identified bad smells for projects JFlex and JFreeChart will be analyzed next. The results for the *Long Method* bad smell will be displayed visually to help the understanding of the categories that have been listed. All other results will be summarized in tabular format.

Figure 4 displays the *Long Method* bad smells that have been identified in the examined versions of JFlex. Each smell is again represented as a horizontal bar spanning across the versions in which the smell is present. (This figure can be regarded as a more detailed representation of what is shown in Figure 3). A line before or after the bar means that the method in which the smell resides existed before the introduction of the smell or after its removal, respectively. In this diagram bad smells corresponding to category A (i.e. smells that exist throughout all versions) have been omitted to improve clarity. All other categories which are present in JFlex are annotated in the Figure along with their frequency.

As already mentioned, the majority of bad smells, once they appear, extend up to the latest version of the system. These smells are the ones corresponding to categories A (not shown in Figure 4) and B which constitute 90% of all cases. This is a clear sign that non-trivial smells are not being removed during the course of evolution as a side effect of usual adaptive and corrective maintenance. The second striking observation is that very few smells disappear in a version during the course of the project (category C and D, 10%). However, as already mentioned, the cases that can be considered as successful smell removal are only the ones corresponding to cases C_1 and D_3 (3.33%), since for case C_2 the problem vanishes only when the method in which it resides is also removed from the project.

Careful examination of the code for the *Long Method* bad smell indicates that none of the few C_1 and D_3 cases can be regarded as a typical, by-the-book application of any of Fowler's refactorings. In other words, for the particular problems that have been identified in this frame of versions, the designers did not extract any code fragment of a method suffering from *Long Method* into a new method, which according to Fowler [7] is the treatment of choice. (The first bar corresponding to the C_2 category appears to be interrupted in one version and then continues up to the end. The reason is that the method in which the smell was located was removed from the code base in that version and re-introduced – under a different name – in the next version. This case could also be classified under the A or B_2 categories but here emphasis is given to the non-intentional removal of the smell).
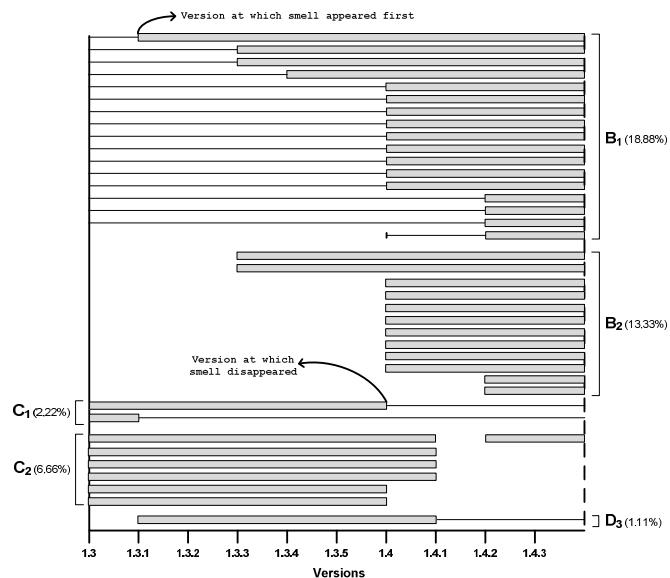

Figure 4. Evolution of *Long Method* bad smells in project JFlex (detailed)

For the second project that has been examined, JFreeChart, due to the large number of identified *Long Method* bad smells it is not possible to present a detailed diagram showing each smell separately. For this reason we present the corresponding categories of bad smells in the schematic of Figure 5, where the width of each bar corresponds to the relative frequency of each category (the corresponding frequency is also shown).

The results for JFreeChart strengthen the previous observation since the problems extending up to the last version correspond to 78.8% of all cases (A+B), implying that *Long Method* smells accumulate with time.
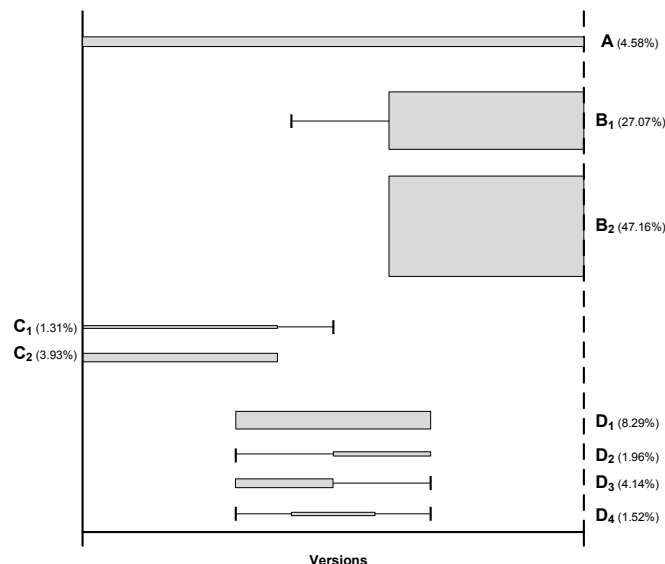

Figure 5. Evolution of *Long Method* bad smells in project JFreeChart

A very large percentage ($B_2+D_1+D_3$, 59.59%) of the cases correspond to design problems that exist right from the beginning of the method in which they reside, implying that the smell was introduced during the initial design/implementation. The corresponding percentage for JFlex was also significant (14.44%). This observation, if verified by other case studies, means that design problems are not only the result of software ageing [20] but also a direct consequence of inefficient initial analysis and design activities.

The cases corresponding to an explicit removal of smells in JFreeChart (C_1, D_3 and D_4) are again limited, while the inspection of the code revealed only three cases with the characteristics of an unambiguous *Extract Method* refactoring, targeting at the separation of functionality into a new method. In all other cases (C_2, D_1 and D_2), which are also limited, the smell was removed when the corresponding method was taken out of the system.

Table II summarizes the findings for all identified bad smell categories, for both projects and all three smells. Data are provided both as absolute numbers as well as percentages. Regardless of the smell frequency it is evident that most smells, once they show up in a version they persist up to the latest examined version (categories $A+B_1+B_2$ constitute on average 74.38% of all cases). On the contrary, the cases where an action (deliberate or not) removed the smell from the system – without removing the containing method or class - ($C_1+D_3+D_4$) are significantly fewer (on average 14.81% of all cases). Concerning the initial appearance of the smells, on average, in 40.33% of all cases ($B_2+D_1+D_3$) the design problem existed when the corresponding method was introduced.

TABLE II.　　IDENTIFIED BAD SMELL CASES

Smell Categories	JFlex			JFreeChart		
	Long Method	*Feature Envy*	*State Checking*	*Long Method*	*Feature Envy*	*State Checking*
A	52 57.77%	8 36.36%	3 60.0%	21 4.58%		7 14.0%
B_1	17 18.88%	3 13.63%	1 20.0%	124 27.07%	5 21.73%	5 10.0%
B_2	12 13.33%	5 22.72%	1 20.0%	216 47.16%	3 13.04%	23 46.0%
C_1	2 2.22%	5 22.72%		6 1.31%		3 6.0%
C_2	6 6.66%	1 4.54%		18 3.93%	1 4.34%	5 10.0%
D_1				38 8.29%	1 4.34%	6 12.0%
D_2				9 1.96%	2 8.69%	
D_3	1 1.11%			19 4.14%	11 47.82%	1 2.0%
D_4				7 1.52%		
Total Number of Distinct Smell Cases	90	22	5	458	23	50

Obviously, the average time of persistence of a bad smell in the system (i.e. for how many of the examined versions it exists) depends on the version that it appeared first and on whether the smell was removed or not. Table III shows the average time of persistence for smells belonging to the three

types that have been examined. A value of 100% would indicate that the average smell of that type exists throughout all examined versions. The relatively high percentages signify that the problems linger on, until gaining the attention of the design team. The fact that some smells, such as *Long Method*, are more common than others (section IV.A) combined with their long persistence, might be used as an indicator to the development team that they warrant much more investment and attention.

TABLE III.　　AVERAGE TIME OF PERSISTENCE

JFlex			JFreeChart		
Long Method	*Feature Envy*	*State Checking*	*Long Method*	*Feature Envy*	*State Checking*
77%	68%	68%	40%	28%	57%

Table IV shows the percentage of unambiguously identified refactorings that have been applied to remove the corresponding bad smell, over all bad smell categories that have been identified for each project/bad smell. According to the collected data, designers do not perform refactorings to remove these three types of design problems. Out of 648 cases of bad smells in total, only in 5 of them a refactoring activity to remove the corresponding smell was undertaken.

TABLE IV.　　UNAMBIGUOUS REFACTORINGS TO REMOVE SMELLS

JFlex			JFreeChart		
Long Method	*Feature Envy*	*State Checking*	*Long Method*	*Feature Envy*	*State Checking*
0 (0.00%)	1 (4.54%)	0 (0.00%)	3 (0.65%)	1 (4.34%)	0 (0.00%)

* Absolute numbers correspond to the identified refactorings. Percentages indicate the ratio of cases where a refactoring was applied over all identified bad smell categories for that project.

Given that other studies [18] have found that refactoring activities are frequent, the findings of our study could possibly mean that designers perform refactorings routinely based on their subjective perception of problematic code areas rather than applying them as solutions to identified design problems. This could be also related to the fact that currently, CASE tools offer support for executing refactorings but have only a limited ability to automatically identify non-trivial bad smells.

The results regarding all three types of bad smells, for all examined versions and both projects are available at [1].

D. Active Bad Smells

A reasonable concern regarding any approach that aims at the identification of code smells or design problems in general is that the identified problems might not seem too worrying for the developers. In that case, it does not come as a surprise if refactoring actions are not taken. As an example, why would it be urgent to improve a fragment of code suffering from the *Long Method* bad smell, if the corresponding method had never been the subject of maintenance? The problem could certainly exist, however among several refactoring opportunities, a suggestion concerning a piece of code that has not been modified in the past, would be possibly ranked lower in the sense that it is not urgent to refactor this aspect of the design.

One of the alternatives to extract information concerning the urgency of a certain refactoring is to employ past versions

of the code. The underlying philosophy is based on the assumption (which of course does not always hold) that code fragments which have been subject to maintenance tasks in the past, are more likely to undergo changes in a future version and thus refactorings involving the corresponding code should have a higher priority. Conversely, if a piece of code remains unmodified over a number of generations, it would not be a top priority for the designer to apply a refactoring affecting it.

To investigate this issue we employ the term "active smell" to refer to a problem where the affected piece of code has been the subject of maintenance, at least once during its history. (The definition stems from volcanology, where according to some researchers an active volcano is one that has erupted some time during its history). It should be mentioned that if the goal is to rank refactoring suggestions, a more sophisticated approach could be used, by assessing for example the extent of modifications or the proximity of past changes to the current version of a system [9].

Concerning *Long Method* smells, the presence of the problem implies that it might be difficult in terms of effort and time to perform maintenance tasks on this method. From this perspective, it makes sense to refactor a method suffering from this design problem, only if we expect that the method will be subject to change in subsequent versions of the system. This means, that previous versions of a system under study should be examined in order to detect changes in the implementation of that particular method. In our analysis, we consider as change of a method between two successive versions, the introduction of new statements, the modification or the removal of existing statements. Even if only one of the three cases occurs for a pair of successive system versions, we record the existence of a change and thus tag any *Long Method* smell concerning that method as active.

Feature Envy problems (in the context of this work) are related to the access of foreign members (attributes and methods). If for a given method, the number of accesses to foreign members remains unaltered during evolution this probably can be interpreted as rather weak evidence of the problem. In other words, if a method uses data or methods from other classes but the number of corresponding statements is not changing then the problem is not as urgent as other cases where the number of accesses changes. Therefore, we tag a *Feature Envy* smell as active if for the corresponding method, the total number of accesses to members of the target class changed at least once during the history of the method.

State Checking smells imply a missed opportunity for employing polymorphism. However, polymorphism makes sense if we expect changes otherwise it introduces needless complexity [14]. Therefore, we tag a *State Checking* smell as active if any of the following has occurred at least once during the examined history of the project:

- addition of new branches in the *if* or *switch* statement on which the smell had been identified (this modification is equivalent to an extension on the underlying axis of change and implies that new subclasses would be added to the introduced inheritance hierarchy)

- change in the number of *State Checking* occurrences related to the same group of implicit states. This is equivalent to the number of times that the introduced hierarchy (if the refactoring were applied) would be used throughout the system (such a modification implies that new fragments of code suffering from the same smell have been added to the system)

- change in the number of statements in the branches of the *if* or *switch* statement on which the smell had been identified (such a modification implies that more code would be moved to the subclasses of the introduced inheritance hierarchy)

Figure 6 shows the identified *State Checking* smells for project JFreeChart and indicates the active ones (shown as grey bars). Moreover, the versions in which any of the aforementioned changes has occurred are indicated by a Greek Delta (in analogy to formal approaches where a Greek Delta implies that the decorated concept undergoes a change). The corresponding symbol is placed in the midway between two versions since changes occur on the transition from one version to the next. As it can be observed, many smells are active, which means that one or more aspects related to the missing use of polymorphism have changed during the evolution of the project. For the *State Checking* symptom, the historical data clearly indicate that most refactoring suggestions are meaningful and the removal of the smell would certainly facilitate maintenance: if polymorphism had been used, none of the recorded changes would impact existing code, reducing the required effort and limiting the possibility of introducing errors. However, it should be again emphasized, that smells which are not tagged as active, are still design problems according to the detection approach; however, their removal is not considered equally urgent according to past changes.
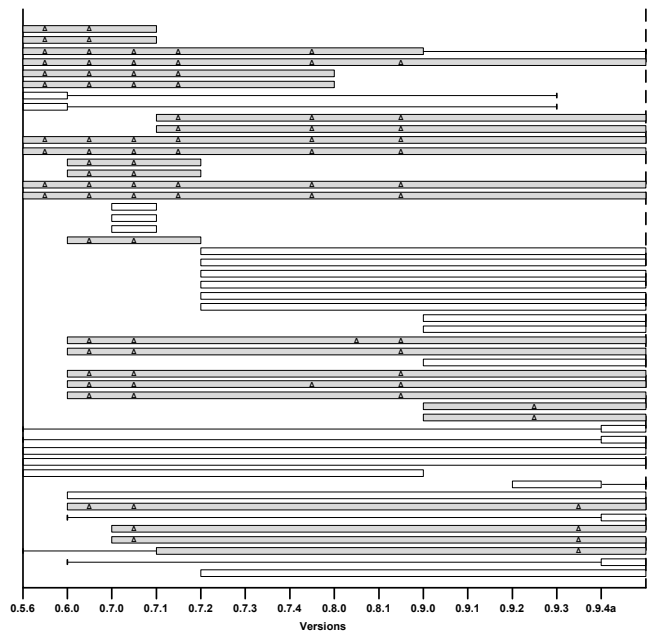

Figure 6. Active *State Checking* bad smells in project JFreeChart

Table V shows the number and percentage of active bad smells, over all bad smell categories that have been identified for each project. If the assumption about the importance of past changes is valid, then these results indicate that a significantly smaller number of smells is alarming. Once again, *Long Method* smells appear to be the most worrying. The larger percentage of active problems for this smell, combined with their larger total number and longer persistence during the history of the projects, implies that maintenance effort should prioritize them over other smells. The uncovering of trends about evolutionary characteristics in order to assist maintenance is exactly one of the major premises of mining past data.

TABLE V.   ACTIVE BAD SMELLS

JFlex			JFreeChart		
Long Method	*Feature Envy*	*State Checking*	*Long Method*	*Feature Envy*	*State Checking*
53	6	1	285	0	26
58.89%	27.27%	20%	62.23%	0%	52%

V.   THREATS TO VALIDITY

Since the case study has been performed employing two projects and three bad smells, the analysis suffers from the usual threats to external validity. In other words, these factors limit the possibility of generalizing our findings beyond the selected setting (projects and smells) and further empirical results are required to strengthen the aforementioned observations.

Two other threats are related to the results of the bad smell identification approach: a) The employed tool may have identified refactoring opportunities which would not be acceptable by a human expert, i.e. smells that are not considered as actual design problems. If such refactoring suggestions exist, it is absolutely reasonable that no refactoring activity was performed to resolve the corresponding problems. b) There might exist refactoring opportunities (or bad smells) which have not been detected by the tool, because they require a different approach in order to be identified.

Finally, another possible threat to construct validity is related to the correct identification of intentional refactoring activities as opposed to code rewriting that resulted in smell removal. However, considering that in most cases, code rewriting causes a change in the behavior, whereas refactorings are behavior-preserving, this distinction is rather clear.

VI.   CONCLUSIONS AND FUTURE WORK

In this paper we presented the results of a case study that investigates the evolution of three bad smells throughout successive versions of two open-source systems. The results indicate that in most cases, the design problems persist up to the latest examined version accumulating as the project matures. Moreover, a significant percentage of the problems were introduced at the time when the method in which they

reside was added to the system. Very few bad smells are removed from the project and in the vast majority of these cases their disappearance was not the result of targeted refactoring activities but rather a side effect of adaptive maintenance.

Future work can be directed to a comparison of bad smells with the results of tools that identify applied refactorings. In this way we can further investigate whether developers perform refactorings that do not correspond to detected smells. Moreover, further statistical analysis could reveal whether certain types of smells tend to be removed quicker than others and the way that refactoring activity is related to the intensity of the underlying problems.

REFERENCES

[1] Bad Smell Evolution Results, http://eos.uom.gr/~achat/SmellResults.rar

[2] Borland Together, http://www.borland.com/together, 2010.

[3] W. H. Brown, R. C. Malveau, H.W. McCormick, and T. J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, Wiley, 1998.

[4] S. Demeyer, S. Ducasse and O. Nierstrasz. "Finding Refactorings via Change Metrics", 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'2000), Minneapolis, USA, October 2000, pp. 166-177.

[5] D. Dig, C. Comertoglu, D. Marinov and R. Johnson, "Automated Detection of Refactorings in Evolving Components", 20th European Conference on Object-Oriented Programming (ECOOP'06), Nantes, France, July 2006, pp. 404-428.

[6] M. Di Penta, L. Cerulo and L. Aversano, "The life and death of statically detected vulnerabilities: An empirical study", Information and Software Technology, vol. 51, issue 10, pp. 1469-1484, October 2009.

[7] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison Wesley, USA, 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

[9] T. Gîrba, S. Ducasse, and M. Lanza, "Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes", 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, USA, September 2004, pp. 40-49.

[10] JDeodorant Eclipse plug-in, http://jdeodorant.com/, 2010.

[11] H. Kagdi, M. L. Collard and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution", Journal of Software Maintenance and Evolution: Research and Practice, vol. 19, issue 2, pp. 77-131, March 2007.

[12] F. Khomh, M. Di Penta and Y.-G. Guéhéneuc, "An Exploratory Study of the Impact of Code Smells on Software Change-proneness", 16th Working Conference on Reverse Engineering (WCRE'09), Lille, France, October 2009, pp. 75-84.

[13] R. Marinescu, "Detecting Design Flaws via Metrics in Object-Oriented Systems", 39th Int. Conf. and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS'01), Santa Barbara, USA, August 2001.

[14] R. C. Martin, Agile Software Development: Principles, Patterns and Practices, Prentice Hall, 2003.

[15] T. M. Meyers and D. Binkley, "An Empirical Study of Slice-Based Cohesion and Coupling Metrics," ACM Transactions on Software Engineering and Methodology, vol. 17, no. 1, pp. 1-27, December 2007.

[16] N. Moha, "Detection and Correction of Design Defects in Object-Oriented Designs", Doctoral Symposiums, Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems, languages and applications (OOPSLA'07), Montreal, Canada, October 2007, pp. 949-950.

[17] 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, Jan./Feb. 2010.

[18] E. Murphy-Hill, C. Parnin, A. P. Black, "How We Refactor, and How We Know It", 31st IEEE Int. Conf. on Software Engineering (ICSE'09), Vancouver, Canada, May 2009, pp. 287-297.

[19] 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", 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'09), Florida, USA, October 2009, pp. 390-400.

[20] D. L. Parnas, "Software aging", 16th Int. Conf. on Software Engineering (ICSE'94), Sorrento, Italy, May 1994, pp. 279–287.

[21] J. Ratzinger, T. Sigmund, and H. C. Gall, "On the relation of refactorings and software defect prediction", 5th Working Conference on Mining Software Repositories (MSR'2008), Leipzig, Germany, May 2008, pp. 35–38.

[22] A. J. Riel, Object-oriented design heuristics, Addison-Wesley, 1996.

[23] S. Slinger, "Code Smell Detection in Eclipse", Thesis Report, Delft University of Technology, Department of Software Technology, March 2005.

[24] A. Trifu, R. Marinescu, "Diagnosing Design Problems in Object Oriented Systems", 12th Working Conference on Reverse Engineering (WCRE'05), Pittsburgh, PA, November 2005, pp. 155-164.

[25] N. Tsantalis and A. Chatzigeorgiou, "Identification of Extract Method Refactoring Opportunities", 13th European Conference on Software Maintenance and Reengineering (CSMR'09), Kaiserslautern, Germany, March 2009, pp. 119-128.

[26] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities", IEEE Transactions on Software Engineering, vol. 35, no. 3, pp. 347-367, May/June 2009.

[27] N. Tsantalis and A. Chatzigeorgiou, "Identification of Refactoring Opportunities Introducing Polymorphism", Journal of Systems and Software, vol. 83, issue 3, pp. 391-404, March 2010.

[28] E. Van Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells", 9th Working Conference on Reverse Engineering (WCRE'02), Richmond, VA, October 2002.

[29] Z. Xing and E. Stroulia, "Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study", 22nd IEEE Int. Conf. on Software Maintenance (ICSM'06), Philadelphia, PA, September 2006, pp. 458-468.

[30] Z. Xing and E. Stroulia, "Refactoring Detection based on UMLDiff Change-Facts Queries", 13th Working Conference on Reverse Engineering (WCRE'06), Benevento, Italy, October 2006, pp. 263-274.