Innovations Syst Softw Eng
DOI 10.1007/s11334-013-0205-z

SI: QUATIC 2010

Investigating the evolution of code smells in object-oriented systems

Alexander Chatzigeorgiou · Anastasios Manakos

Received: 29 June 2011 / Accepted: 6 April 2013

Abstract Software design problems are known and perceived under many different terms, such as code smells, flaws, non-compliance to design principles, violation of heuristics, excessive metric values and anti-patterns, 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 code 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 code 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 analyze the evolution from the point of view of the problems themselves. To this end, we classify smell evolution patterns 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 four code smells.

Keywords Code smell · Refactoring · Software repositories · Software history · Evolution

A. Chatzigeorgiou (✉) · A. Manakos
Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
e-mail: achat@uom.gr

A. Manakos
e-mail: mai0932@uom.gr

1 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 [27]. Declining quality of evolving systems is also something that is expected according to Lehman's 7th law of software evolution [18]. The importance that the software engineering community places on the detection and resolution of design problems is evident from the multitude of terms under which they are known. Some researchers view problems as non-compliance with design principles [20], violations of design heuristics [29], excessive metric values, lack of design patterns [12] or even application of anti-patterns [3].

According to Fowler [11], 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. Their popularity is also evident from the availability of numerous tools that provide support for the application of refactorings relieving the designers from the burden of their mechanics [24].

According to the recommendations proposed by Lehman and Ramil for software evolution planning [18], quality should be continuously monitored as systems evolve. This implies that past versions of a software system should be analyzed to track changes in evolutionary trends. To this end, organized collections of software repositories offer an addi-

Published online: 21 April 2013

tional, 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) [16], 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 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 four code smells regarding design issues, by looking at various past versions of two open-source systems. The tool that has been employed is JDeodorant [15] which allows the identification of four non-trivial code smells, namely *Long Method*, *Feature Envy, State Checking* and *God Class*. 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 code smells or are the problems solved only after targeted maintenance activities?
- Do code 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 code smells?
- How long do code smells "survive" inside software systems?
- How urgent is it to remove the identified code smells?

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: Related work on refactoring identification approaches, empirical studies regarding refactoring practice and tools that allow the detection of code smells is presented in Sect. 2. Section 3 describes briefly the essence of the code smells that have been investigated and the overall strategy that JDeodorant uses to detect them. In Sect. 4, results concerning the number, types and evolution patterns of code smells in the examined projects are presented in visual and tabular form and findings are discussed. Section 5 investigates the persistence of smells in source code by means of survival analysis. The notion of active smells along with results for all smells and projects are discussed in Sect. 6. Threats to the validity of the study are listed in Sect. 7. Finally, we conclude in Sect. 8.

This manuscript is an extended version of paper [6] including results for an additional code smell and the survival analysis described in Sect. 5.

2 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. [7] 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. [8] 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 [38]. 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 to investigate whether refactorings are performed regularly and systematically and to explore programmers' and maintainers' habits regarding refactoring practice.

Xing and Stroulia [37] 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 code smells.

An extensive study of refactoring application has been presented by Murphy-Hill et al. [25] 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 [25], 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 [28] 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. [25].

Recently, a number of researchers investigated the impact of code smells on change-proneness. Olbrich et al. [26] 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 [17], 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. [9]. 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 code 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 [19] which employs selected metrics to identify suspect classes that might exhibit design problems, such as *God* and data classes, jCOSMO [36] and its successor CodeNose [30] where identification of a code smell is assumed when all associated smell aspects are found using static analysis, iPlasma [31] which uses a detection strategy based on the composition of various metric rules combined with AND/OR operators to express design heuristics, DÉCOR [22,23] 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.

3 Code smells

As already mentioned, this study employs JDeodorant for the identification of code smells. The main reason is that the tool offers the possibility to detect non-trivial code 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 code smell. The four code smells that have been studied are:

3.1 Long Method

Methods suffering from the *Long Method* code 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 [21]). An ideal solution to this kind of design problems is given by the Extract Method refactoring [11] which simplifies the code by breaking large methods into smaller ones and creates new methods which can be reused.

JDeodorant identifies *Long Method* code smells and, in particular, detects refactoring opportunities which (a) extract the complete computation of a given variable into a new method [32], 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.

3.2 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" [11]. 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* code smells as opportunities for Move Method refactoring [33]. 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 [33]. 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.

3.3 State Checking

State Checking (known under the name Switch Statements in [11]) 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 [20] 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* code smells as opportunities for introducing polymorphism [34]. 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.

3.4 God Class

"*God*" classes are large, complex and non-cohesive modules that violate the principle of implementing only one concept per class [20] and are difficult to understand and maintain [29]. In general, two types of *God classes* can be found in object-oriented code: "Data God" classes that house a large percentage of the system's state in terms of number of attributes and "Behavioral God" classes that incorporate a large fraction of the system's functionality in terms of number of methods. This smell can be addressed by refactorings such as "Move Method" and "Move Attribute", which aim at extracting methods and/or attributes to other (new or existing) classes to improve the cohesion of the involved modules.

JDeodorant identifies *God Class* problems as opportunities for extracting cohesive groups of class members (methods and attributes) as separate classes. To this end, it employs a clustering algorithm based on a distance metric derived from the dependencies (i.e. field accesses and method invocations) between class members [10]. The identified clusters are considered as candidates for extraction and are ranked according to the anticipated impact on the design quality, i.e. how much they improve coupling and cohesion. In the context of the current study, every class for which one or more

Table 1 Size characteristics of the examined versions/projects

JFlex:	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				
Measures														
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				
JFreeChart:	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
Measures														
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

extract class opportunities are found is considered as a class suffering from the *God Class* smell.

4 Smell evolution

4.1 Case studies

In the presented empirical 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). Code 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 1.

4.2 Total number of code smells

Since functionality is enhanced in every new version of a software system and since open-source software does not undergo systematic preventive maintenance, it is reasonable to expect that the total number of design smells will increase with time. The results, summarized in Figs. 1 and 2, confirm this belief, for both systems and all four of the selected code 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 three symptoms. In almost all cases, the number of problems increases as the system evolves, although the rate of increase is lower for *Feature Envy, State Checking* and *God Class* smells.

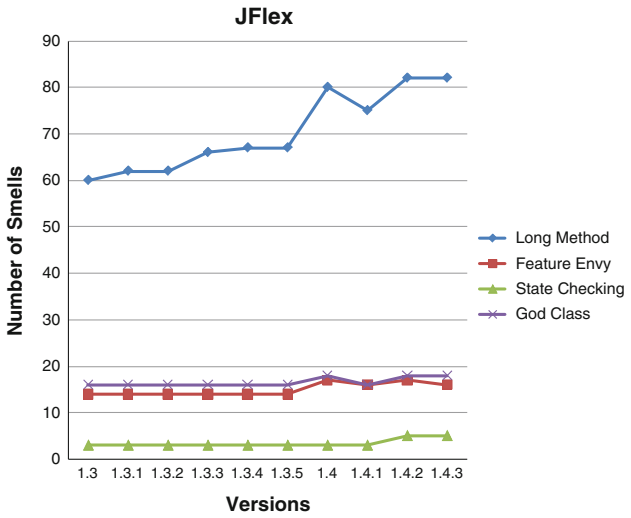

Fig. 1 Total number of smells in project JFlex

4.3 Persistence of code smells

To provide an overview of the way design problems evolve over time, we employ a specific graphical notation. For example, in Fig. 3, we have plotted for project JFlex the way in which *Long Method* code smells spread over successive versions. Each horizontal grey bar corresponds to an identified code 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 code 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

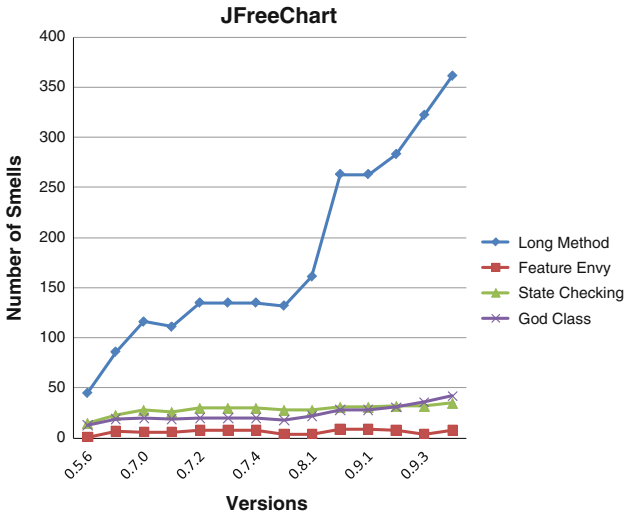

Fig. 2 Total number of smells in project JFreeChart

Fig. 3 Evolution of *Long Method* code smells in project JFlex

the examined versions. (The total number of distinct smell cases for both projects and all four smells is shown in the last row of Table 2.)

Regarding the relatively few cases where the existence of a code 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 code 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* code smell. Thus, we consider these cases as accidental elimination of the smell.
- *Code removal* The entire code fragment suffering from a code 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* code 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. For a *God Class* smell, an unambiguous refactoring is implied if methods and/or attributes (forming a reasonable concept) from the problematic class have been moved to another (new or existing) class. 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, 0.69 % of all cases).

4.4 Evolution patterns of code smells

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

Table 2 Identified code smell cases

Smell categories	JFlex				JFreeChart			
	Long Method	Feature Envy	State Checking	God Class	Long Method	Feature Envy	State Checking	God Class
A	52	8	3	13	21		7	8
	(57.77 %)	(36.36 %)	(60.0 %)	(65.0 %)	(4.58 %)		(14.0 %)	(14.04 %)
B_1	17	3	1	3	124	5	5	16
	(18.88 %)	(13.63 %)	(20.0 %)	(15.0 %)	(27.07 %)	(21.73 %)	(10.0 %)	(28.07 %)
B_2	12	5	1	1	216	3	23	19
	(13.33 %)	(22.72 %)	(20.0 %)	(5.0 %)	(47.16 %)	(13.04 %)	(46.0 %)	(33.33 %)
C_1	2	5		2	6		3	2
	(2.22 %)	(22.72 %)		(10.0 %)	(1.31 %)		(6.0 %)	(3.51 %)
C_2	6	1			18	1	5	1
	(6.66 %)	(4.54 %)			(3.93 %)	(4.34 %)	(10.0 %)	(1.75 %)
D_1					38	1	6	3
					(8.29 %)	(4.34 %)	(12.0 %)	(5.26 %)
D_2					9	2		5
					(1.96 %)	(8.69 %)		(8.77 %)
D_3	1				19	11	1	1
	(1.11 %)				(4.14 %)	(47.82 %)	(2.0 %)	(1.75 %)
D_4				1	7			2
				(5.0 %)	(1.52 %)			(3.51 %)
Total number of distinct smell cases	90	22	5	20	458	23	50	57

– 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:

  – B1: Smells that appeared at a point during the evolution of a project but did not exist when the method/class in which they reside was introduced. These cases imply that the particular design problem was introduced during evolution or maintenance of the method/class under study.
  – B2: Smells that exist right from the beginning of the corresponding method/class, that is, from the point at which the method/class 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.

  – C1: Code smells that have been removed whereas the corresponding method/class 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.
  – C2: Smells that exist right from the first version that has been analyzed and have been eliminated from the project because the corresponding method/class has been removed from the system. Obviously, these cases cannot be considered as successful refactoring applications since the method/class that presented the smell has been completely eliminated.

– D: Code 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/class containing the problem:

  – D1: The smell appeared when the corresponding method/class was introduced. The smell disappeared when the method/class was removed from the system.
  – D2: The smell appeared during the evolution of the method/class (i.e. as a result of its adaptive or corrective maintenance) and disappeared when the method/class was removed.
  – D3: The smell appeared when the corresponding method/class was introduced. The method/class continued to exist after the removal of the smell.
  – D4: The smell appeared and disappeared during the evolution of the method/class (i.e. the method/class

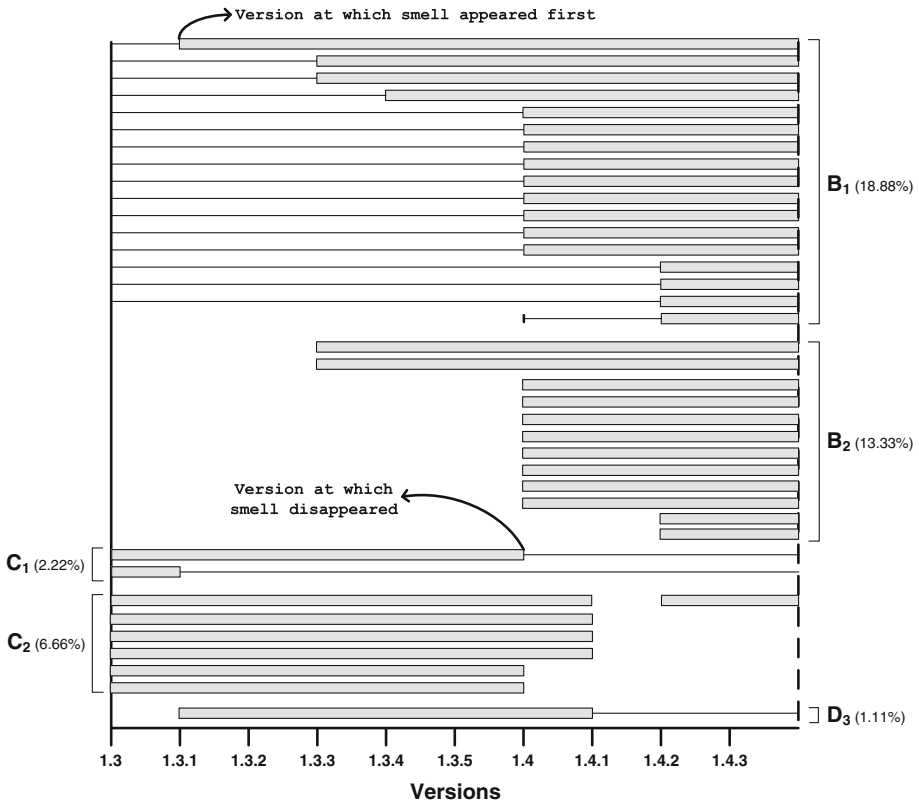

Fig. 4 Evolution of Long Method code smells in project JFlex (detailed)

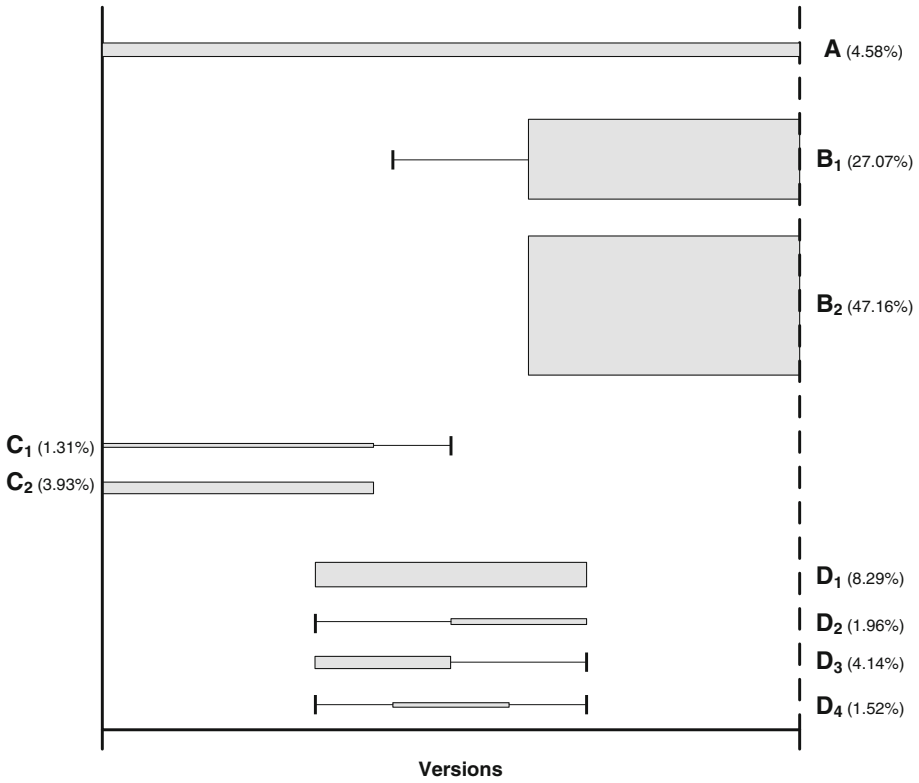

Fig. 5 Evolution of Long Method code smells in project JFreeChart

existed before the introduction of the smell and after its removal). Cases belonging to D3 and D4 categories can potentially be regarded as successful code removal activities.

The results concerning the identified code smells for projects JFlex and JFreeChart will be analyzed next. The results for the *Long Method* code 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* code 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 Fig. 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 code, 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 code 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 Fig. 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 C1 and D3 (3.33 %), since for case C2 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* code smell indicates that none of the few C1 and D3 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 [11] is the treatment of choice. (The first bar corresponding to the C2 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 B2 categories but here emphasis is given to the non-intentional removal of the smell).

For the second project that has been examined, JFreeChart, due to the large number of identified *Long Method* code smells, it is not possible to present a detailed diagram show-

Table 3 Unambiguous refactorings to remove smells

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

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

ing each smell separately. For this reason, we present the corresponding categories of code smells in the schematic of Fig. 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.

A very large percentage (B2 + D1 + D3, 59.59 %) of the cases corresponds 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 [27] but also a direct consequence of inefficient initial analysis and design activities.

The cases corresponding to an explicit removal of smells in JFreeChart (C1, D3 and D4) 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 (C2, D1 and D2), which are also limited, the smell was removed when the corresponding method was taken out of the system.

Table 2 summarizes the findings for all identified code smell categories, for both projects and all four 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, persist up to the latest examined version (categories A + B1 + B2 constitute on average 75.84 % 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—(C1 + D3 + D4) are significantly fewer (on average 14.08 % of all cases). Concerning the initial appearance of the smells, on average, in 35.91 % of all cases (B2 + D1 + D3) the design problem existed when the corresponding method or class was introduced.

To find out in how many cases of smell elimination, the development or maintenance team applied typical refactoring actions to resolve the corresponding problems, the source code for the involved pieces of code has been manually examined for the first problem-free version as well as the immediately previous one. Table 3 shows the percentage of unambiguously identified refactorings that have been applied to remove the corresponding code smell, over all code smell categories that have been identified for that project/code smell. According to the collected data, designers do not perform refactorings to remove these four types of design problems. Out of 725 cases of code smells in total, only in 5 of them a refactoring activity to remove the corresponding smell was undertaken.

Given that other studies [25] 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 code smells.

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

5 Survival analysis

The afore-mentioned results concerning the time point of the introduction and the elimination of code smells during the evolution of software enables the study of how long code smells "survive" inside the corresponding systems. Obviously, the average time of persistence of a code 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. By applying simple descriptive statistics, we obtain the results of Table 4 that shows the average time of persistence for smells belonging to the four types that have been examined (as percentage and as absolute numbers). A value of 100 % indicates that the average smell of that type exists throughout all examined versions. (Absolute numbers correspond to the mean duration of the average smell in numbers of versions.) The relatively high percentages signify that the problems linger

Table 4 Average time of persistence

JFlex				JFreeChart			
Long Method	Feature Envy	State Checking	God Class	Long Method	Feature Envy	State Checking	God Class
77 %	68 %	68 %	79 %	40 %	28 %	57 %	42 %
(7.7)	(6.8)	(6.8)	(7.9)	(5.6)	(3.9)	(8.0)	(5.9)

on, until gaining the attention of the design team. The fact that some smells, such as *Long Method*, are more common than others (Sect. 4.2) combined with their long persistence might be used as an indicator to the development team that they warrant much more investment and attention.

However, by calculating the average time of persistence by simple descriptive statistics, we do not take into account the fact that some smells that are present up to the latest examined version might continue to exist in the subsequent versions beyond the end point of the study. This parameter is considered by survival analysis which is primarily used in biomedical sciences and deals with the investigation of the occurrence of events (such as death, disease recurrence, etc.) over time, when the time-to-event is the parameter of interest [14]. One of the major issues in survival analysis is that during the collection of data for a particular survey, for some observations the critical event might have not been observed yet. For example the subject (e.g. a patient) might have not experienced the event before the study ends or the subject might have left the study [5]. The corresponding observations are said to be censored which means that some information is available regarding the event time, but the exact event time is not known. Obviously, the exclusion of these observations is not a solution since this would bias the results. The goal of survival analysis is to incorporate censored observations and extract unbiased estimates regarding the time it takes for an event to occur [14].

The Kaplan–Meier curve is the most commonly used method to graphically depict an estimate of the survival function measuring the fraction of subjects (such as patients) "living" for a certain amount of time. For each interval of the curve, survival probability is estimated from the cumulative probability of surviving each of the preceding time intervals. Kaplan–Meier curves can be easily interpreted and allow the comparison of time-to-event between two or more groups or treatments.

In the context of code smell evolution, one could consider as critical event the elimination of a smell (regardless of whether it is intentional or not). Survival analysis could then provide information regarding the time it takes for a smell to disappear from the system, in other words for how many successive versions of the examined software a smell "survives" in it. As already mentioned, for most of the examined smells, once the problems are introduced in the system they

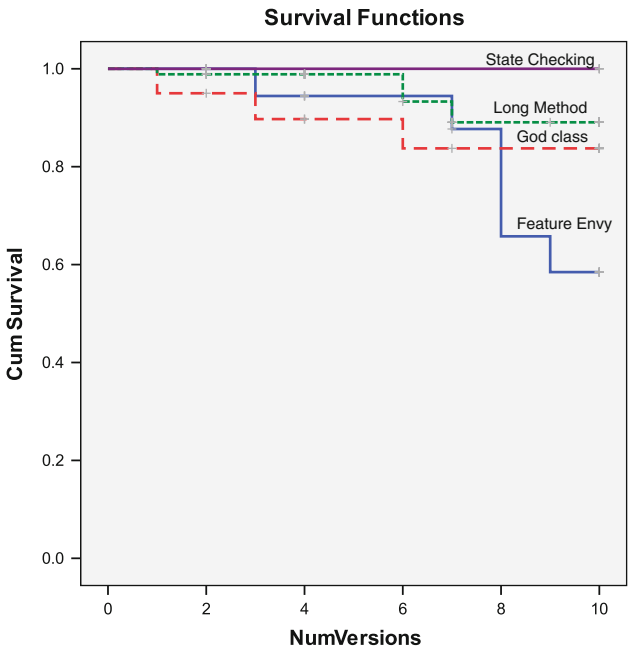

Fig. 6 Kaplan–Meier curves for code smells in project JFlex

persist up to the latest examined version, leading unavoidably to a large number of censored observations. For project JFlex and all four examined smells, the Kaplan–Meier curve showing an estimate for the corresponding survival functions in shown in Fig. 6. (It should be mentioned that all five smell cases for *State Checking* persist up to the latest examined version and as a result all observations are censored).

As it can be readily observed, code smells tend to "survive" for a long time within the examined systems. According to the results, after 10 successive software versions, approximately 60 % of *Feature Envy*, 85 % of *God Class*, 90 % of *Long Method* and 100 % of *State Checking* smells are still present in the system. Feature Envy smells tend to have a relatively shorter time-to-elimination compared to the other three smells. For example, the mean survival time for *Long Method* smells is 9.5 versions while for *Feature Envy* smells is 8.9 versions.

The Kaplan–Meier curve for project JFreeChart is shown in Fig. 7. In this case, the time-to-elimination is shorter than in project JFlex, but after 10 versions a large percentage (>60 %) of *Long Method*, *State Checking* and *God Class*

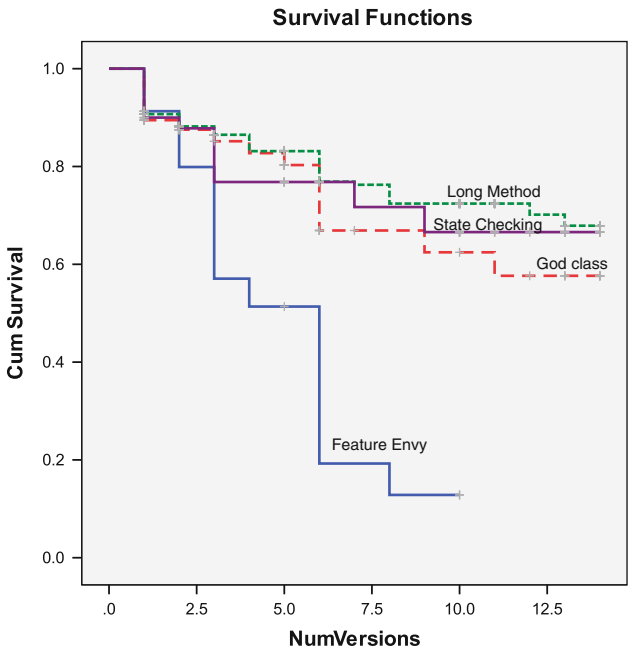

Fig. 7 Kaplan–Meier curves for code smells in project JFreeChart

smells continue to exist in the code. A remarkable difference can be observed for *Feature Envy*, where after 8 versions only a very small percentage of smells (approximately 10 %) remained in the system. The mean survival time for *Long Method* smells is 11.1 versions while for *Feature Envy* smells is 4.9 versions.

The mean survival time (in number of versions) obtained by Kaplan–Meier analysis for all four examined smells and both projects is given in Table 5. As it can be observed, the time is longer than the average time obtained by simple descriptive statistics, since censored data are appropriately handled. Differences are significant between smells for the same project as well as between the same smell for two different projects. According to the results, *Long Method* smells are the most difficult to get rid of, either by applying intentional refactoring activities or unintentionally by performing usual system maintenance.

6 Active code 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, for most designers, it would not be urgent to improve a fragment of code suffering from the *Long Method* code 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 noted that not all kind of changes should be considered; only modifications to the code involved in the corresponding smells or in other words, only changes that modify the presence or intensity of the smell. If the goal is to rank refactoring suggestions, a more sophisticated approach could be used, by assessing, for example, the frequency of past changes, the extent of modifications or the proximity of past changes to the current version of a system [4,13,35].

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 to detect changes in the implementation of that

Table 5 Mean survival time

	JFlex				JFreeChart			
	Long Method	Feature Envy	State Checking	God Class	Long Method	Feature Envy	State Checking	God Class
Mean time	9.5	8.9	[a]	8.9	11.1	4.9	10.6	10.2
Std. error	0.159	0.431	[a]	0.582	0.266	0.647	0.745	0.766

[a] No statistics can be computed since all cases are censored

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 [20]. 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).

God class symptoms indicate heavily loaded classes that incorporate a large fraction of system's logic or state. They

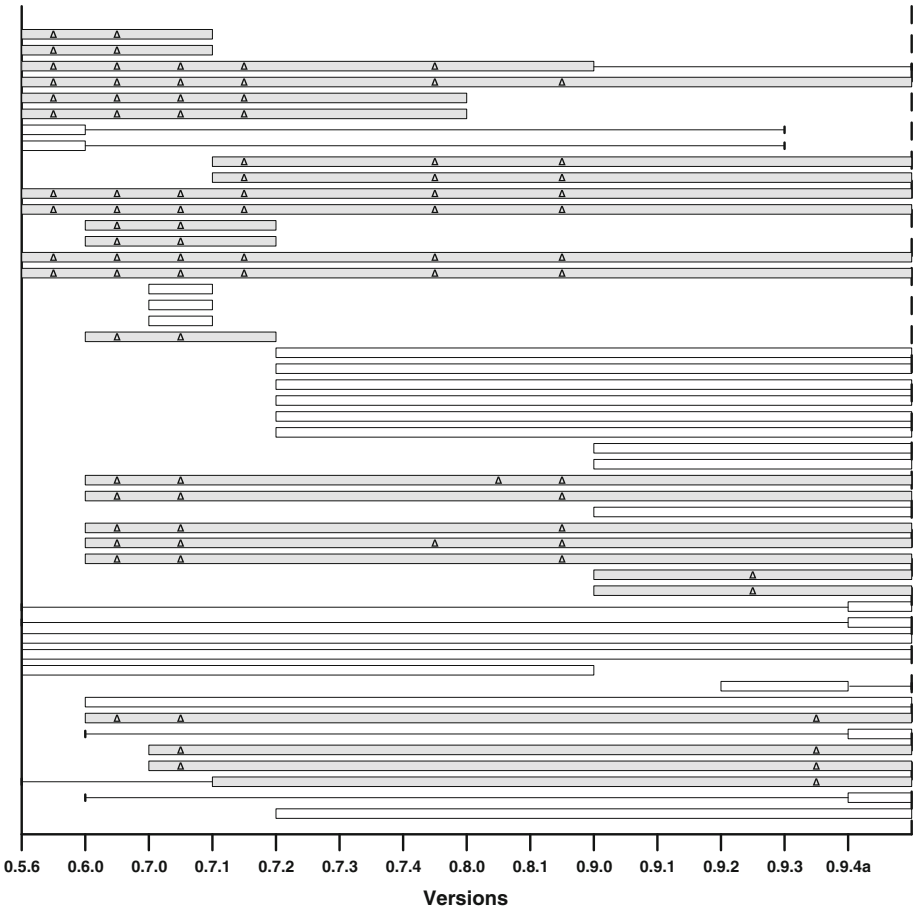

Fig. 8 Active *State Checking* code smells in project JFreeChart

Table 6 Active code smells

JFlex				JFreeChart			
Long Method	Feature Envy	State Checking	God Class	Long Method	Feature Envy	State Checking	God Class
53 (58.89 %)	6 (27.27 %)	1 (20 %)	11 (55 %)	285 (62.23 %)	0 (0 %)	26 (52 %)	25 (43.86 %)

are the result of inefficient and non-uniform allocation of methods and attributes to the system classes. A *God class* problem will become even more severe if we add methods and/or attributes to its entity set and in most cases will remain a *God class,* unless we remove functionality or variables. Obviously, *God classes* which tend to become larger and more complex with the passage of software versions should be assigned a higher priority for resolving the corresponding smells. In this context, a *God class* smell should be considered active, if during the history of the class the total number of methods/attributes of the suffering class changed (increased) in at least one version.

Figure 8 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.

Table 6 shows the number and percentage of active code smells, over all code 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 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 to assist maintenance is exactly one of the major premises of mining past data.

7 Threats to validity

Since the case study has been performed employing two projects and four code 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 afore-mentioned observations.

Two other threats are related to the results of the code 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 code 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.

8 Conclusions

In this paper, we presented results concerning the evolution of four code smells throughout successive versions of two open-source systems. The findings indicate that in most cases, the design problems persist up to the latest examined version accumulating as the project matures. Survival analysis has shown that smells "live" for a large number of versions thus being a permanent problem once they are introduced in software. Moreover, a significant percentage of the problems was introduced at the time when the method or class in which they reside was added to the system. Very few code 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.

A future line of research could be the comparison of detected code smells with the results of tools that identify applied refactorings in past software versions. In this way, we can further investigate whether developers perform refactorings that do not correspond to detected smells. Further empirical analysis on several software systems and other smells including more common design problems would validate whether there is a mapping between refactoring activities and the underlying problems or not.

Acknowledgments This work has been partially funded by the Research Committee of the University of Macedonia, Greece. We would also like to thank the anonymous reviewers of QUATIC'2010 for their constructive comments on the manuscript.

References

1. Bad Smell Evolution Results (2011) http://eos.uom.gr/~achat/SmellResults.rar. Accessed 23 June 2011
2. Borland Together (2011) http://www.borland.com/us/products/together/. Accessed 23 June 2011
3. Brown WH, Malveau RC, McCormick HW, Mowbray TJ (1998) AntiPatterns: refactoring software, architectures, and projects in crisis. Wiley, New York
4. Capiluppi A, Fernández-Ramil J (2007) A model to predict anti-regressive effort in Open Source Software. In: Proceedings of 23rd international conference on software maintenance (ICSM'2007), Paris, France, pp 194–203
5. Chan YH (2004) Biostatistics 203. Survival analysis. Singap Med J 45:249–256
6. Chatzigeorgiou A, Manakos A (2010) Investigating the evolution of bad smells in object-oriented code. In: Proceedings of 7th international conference on the quality of information and communications technology (QUATIC'2010), Porto, Portugal, pp 106–115
7. Demeyer S, Ducasse S, Nierstrasz O (2000) Finding refactorings via change metrics. In: Proceedings of 15th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications (OOPSLA'2000), Minneapolis, USA, pp 166–177
8. Dig D, Comertoglu C, Marinov D, Johnson R (2006) Automated detection of refactorings in evolving components. In: Proceedings of 20th European conference on object-oriented programming (ECOOP'06), Nantes, France, pp 404–428
9. Di Penta M, Cerulo L, Aversano L (2009) The life and death of statically detected vulnerabilities: an empirical study. Inf Softw Technol 51:1469–1484
10. Fokaefs M, Tsantalis N, Chatzigeorgiou A, Sander J (2009) Decomposing object-oriented class modules using an agglomerative clustering technique. In: Proceedings of 25th IEEE international conference on software maintenance (ICSM'2009), Edmonton, Alberta, Canada, pp 93–101
11. Fowler M (1999) Refactoring: improving the design of existing code. Addison Wesley, Boston
12. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison Wesley, Boston
13. Gîrba T, Ducasse S, Lanza M (2004) Yesterday's weather: guiding early reverse engineering efforts by summarizing the evolution of changes. In: Proceedings of 20th IEEE international conference on software maintenance (ICSM'04), Chicago, USA, pp 40–49
14. Hox JJ (2002) Multilevel analysis: techniques and applications. Routledge Academic, London
15. JDeodorant Eclipse plug-in (2011) http://jdeodorant.com/ Accessed 23 June 2011
16. Kagdi H, Collard ML, Maletic JI (2007) A survey and taxonomy of approaches for mining software repositories in the context of software evolution. J Softw Main Evol R 19:77–131
17. Khomh F, Di Penta M, Guéhéneuc YG (2009) An exploratory study of the impact of code smells on software change-proneness. In: Proceedings of 16th working conference on reverse engineering (WCRE'09), Lille, France, pp 75–84
18. Lehman MM, Ramil JF (2001) Rules and tools for software evolution planning and management. Ann Softw Eng 11:15–44
19. Marinescu R (2001) Detecting design flaws via metrics in object-oriented systems. In: Proceedings of 39th international conference and exhibition on technology of object-oriented languages and systems (TOOLS'01), Santa Barbara, USA
20. Martin RC (2003) Agile software development: principles, patterns and practices. Prentice Hall, New Jersey
21. Meyers TM, Binkley D (2007) An empirical study of slice-based cohesion and coupling metrics. ACM Trans Softw Eng Methodol 17:1–27
22. Moha N (2007) Detection and correction of design defects in object-oriented designs. In: Proceedings of 22nd ACM SIGPLAN conference on object-oriented programming systems, languages and applications (OOPSLA'07), Doctoral Symposiums, Montreal, Canada, pp 949–950
23. Moha N, Guéhéneuc YG, Duchien L, Le Meur AF (2010) DECOR: a method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36:20–36
24. Murphy-Hill E, Black AP (2008) Refactoring tools: fitness for purpose. IEEE Softw 25:38–44
25. Murphy-Hill E, Parnin C, Black AP (2009) How we refactor, and how we know it. In: Proceedings of 31st IEEE international conference on software engineering (ICSE'09), Vancouver, Canada, pp 287–297
26. Olbrich S, Cruzes DS, Basili V, Zazworka N (2009) The evolution and impact of code smells: a case study of two open source systems. In: Proceedings of 3rd international symposium on empirical software engineering and measurement (ESEM'09), Florida, USA, pp 390–400
27. Parnas DL (1994) Software aging. In: Proceedings of 16th international conference on software engineering (ICSE'94), Sorrento, Italy, pp 279–287
28. Ratzinger J, Sigmund T, Gall HC (2008) On the relation of refactorings and software defect prediction. In: Proceedings of 5th working conference on mining software repositories (MSR'2008), Leipzig, Germany, pp 35–38
29. Riel AJ (1996) Object-oriented design heuristics. Addison-Wesley, Boston
30. Slinger S (2005) Code smell detection in eclipse. Dissertation, Delft University of Technology
31. Trifu A, Marinescu R (2005) Diagnosing design problems in object oriented systems. In: Proceedings of 12th working conference on reverse engineering (WCRE'05), Pittsburgh, PA, pp 155–164
32. Tsantalis N, Chatzigeorgiou A (2009) Identification of extract method refactoring opportunities. In: Proceedings of 13th European conference on software maintenance and reengineering (CSMR'09), Kaiserslautern, Germany, pp 119–128
33. Tsantalis N, Chatzigeorgiou A (2009) Identification of move method refactoring opportunities. IEEE Trans Softw Eng 35: 347–367
34. Tsantalis N, Chatzigeorgiou A (2010) Identification of refactoring opportunities introducing polymorphism. J Syst Softw 83:391–404
35. Tsantalis N, Chatzigeorgiou A (2011) Ranking refactoring suggestions based on historical volatility. In: Proceedings of 15th European conference on software maintenance and reengineering (CSMR'2011), Oldenburg, Germany, pp 25–34

36. Van Emden E, Moonen L (2002) Java quality assurance by detecting code smells. In: Proceedings of 9th working conference on reverse engineering (WCRE'02), Richmond, VA, pp 97–106
37. Xing Z, Stroulia E (2006) Refactoring practice: how it is and how it should be supported—an eclipse case study. In: Proceedings of 22nd IEEE international conference on software maintenance (ICSM'06), Philadelphia, PA, pp 458–468
38. Xing Z, Stroulia E (2006) Refactoring detection based on UMLDiff change-facts queries. In: Proceedings of 13th working conference on reverse engineering (WCRE'06), Benevento, Italy, pp 263–274