Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 820103, 12 pages
doi:10.1155/2010/820103

Research Article

Exploring the Eradication of Code Smells: An Empirical and Theoretical Perspective

S. Counsell,[1] R. M. Hierons,[1] H. Hamza,[1] S. Black,[2] and M. Durrand[3]

[1] *Department of IS and Computing, Brunel University, Uxbridge UB8 3PH, UK*
[2] *Department of Information and Software Systems, University of Westminster, Harrow Campus, London HA1 4TP, UK*
[3] *change-s.com, Westminster Borough, London, UK*

Correspondence should be addressed to S. Counsell, steve.counsell@brunel.ac.uk

Received 2 September 2010; Revised 31 December 2010; Accepted 31 December 2010

Academic Editor: Giulio Concas

Code smells reflect code decay, and, as such, developers should seek to eradicate such smells through application of "deodorant" in the form of one or more refactorings. However, a relative lack of studies exploring code smells either theoretically or empirically when compared with literature on refactoring suggests that there are reasons why smell eradication is neither being applied in anger, nor the subject of significant research. In this paper, we present three studies as supporting evidence for this stance. The first is an analysis of a set of five, open-source Java systems in which we show very little tendency for smells to be eradicated by developers; the second is an empirical study of a subsystem of a proprietary, C# web-based application where practical problems arise in smell identification and the third, a theoretical enumeration of smell-related refactorings to suggest why smells may be left alone from an effort perspective. Key findings of the study were that first, smells requiring application of simple refactorings were eradicated in favour of smells requiring more complex refactorings; second, a wide range of conflicts and anomalies soon emerged when trying to identify smelly code; an interesting result with respect to comment lines was also observed. Finally, perceived (estimated) effort to eradicate a smell may be a key factor in explaining why smell eradication is avoided by developers. The study thus highlights the need for a clearer research strategy on the issue of code smells and all aspects of their identification and measurement.

1. Introduction

Bad code smells are a concept described in Fowler [1] to denote code that "screams out" to be refactored. In other words, it reflects code that is decaying and, unless eradicated, is likely to be the cause of frequent future maintenance, potential faults, and associated testing overheads. Left unchecked and free to fester over time, smells can soon become "stenches" with the potential for relatively high fault-proneness, added maintenance demands, and extratesting as outcomes. An evolving system left to decay is a problem "stored-up" and growing in size for a later date. Eradication of a smell is usually achieved through application of a single, or set of, refactoring/s, and twenty-two different code smells are described by Fowler in [1] together with the refactorings that are needed to remedy those smells. While the related topic of refactoring has been the subject of a significant number of research and other studies [2–9], the empirical and theoretical study of code smells seems to have been largely overlooked by the research community. Even the terminology used in code smell research has yet to find a firm footing and general acceptance. And yet, the problem of code smells has strong industrial resonance—decaying systems consume vast developer resources.

In this paper, we describe three studies of supporting evidence to justify our stance that a fresh look needs to be made of the area, the scope for research in the area, and the benefits that analysis of code smells could provide. The first presents a "smell-to-refactoring" theoretical justification for why some smells may be prohibitive for developers to remedy based on the number of related refactorings required to eradicate a smell; in other words, we suggest that the necessary effort required for smell eradication might itself be prohibiting factor. The second is a study of multiple versions

of five, Java open-source systems (OSSs) [2] from which extracted refactorings, when reverse engineered, showed little empirical propensity on the part of the OSS developers to eradicate smells. Finally, we describe an empirical examination of two versions of a proprietary C#, web-based system in which we point to why even identifying simple smells such as "large" classes and "long" methods [1] pose practical difficulties and raise both conflicts and anomalies. Results showed that perceived (estimated) effort to eradicate a smell may be a key factor in explaining why smell eradication is avoided by developers. Only limited evidence of smell eradication by developers and a wide range of practical problems soon emerged when trying to identify "smelly" code from proprietary C# code. The remainder of the paper is organized as follows. In Section 2, we present the motivation for the work. We then present each of the three studies in the order described with supporting data (Sections 3, 4, and 5, resp.) and describe threats to study validity in Section 6. We then finalize with conclusions and future work (Section 7).

2. Motivation/Related Work

The research in this paper is motivated by one overarching research question: why, if the eradication of code smells provides such obvious potential, theoretical benefits and is a problem that all code might suffer from as it evolves, has the same topic received such little research attention? This question itself induces a range of other questions and motivating factors. First, what role does human judgment and motivation for eradicating smells fulfill in the identification of smells? In other words, are developers interested in smell eradication? It is clear that the choice of what a "large" class or "excessive" coupling constitutes is subjective and this might lie at the heart of why developers are reluctant to address code smells. Second, what anomalies and inconsistencies arise when we attempt to "sniff out" smells from systems? That is, if we consciously search out code smells, what practical problems arise? Third, we need to consider the opportunity cost of choosing to eradicate one code smell over another. Developer time is limited, and there is a high opportunity cost of any smell eradication effort. Fourth, what theoretical considerations become important for the practical eradication of code smells? The activity of eradicating a single smell can, in theory, require a range of subactivities, depending on smell "complexity." Finally, we cannot discount from our discussion the burden that increased testing places on the developer. Just as when a developer undertakes a simple refactoring, for every smell eradicated there is a need to test the resulting code to ensure it has retained its "semantics"; eradication of a smell poses a similar challenge.

In terms of related work, research into code smells started promisingly with several industry-oriented studies, but seems to have petered out more recently. While smells are widely acknowledged as a problematic aspect of software development, very little research work has focused on code smells, their analysis and even less on the empirical study of smells from industrial, proprietary code. Two notable, seminal studies of code smells were undertaken by Mäntylä et al. [10] and Mäntylä and Lassenius [11, 12] who conducted an empirical study of industrial developers and their opinion of smells in evolving structures. The study gave insights into which smells developers most "understood" and hence they would be most likely to eradicate—the "Large Class" smell [1] featured prominently. A well-known "taxonomy" for allocating code smell was also proposed by Mäntylä in [13]; in subsequent work, Mäntylä and Lassenius also describe mechanisms for making refactoring decisions based on smell identification [11]. Recent research by Khomh et al. explored code smells using a Bayesian network approach [14] and from looking at changes made to a system as a basis for smell identification [15]. Counsell et al. established a link between refactoring and code smells in terms of the in- and out-degrees of a dependency diagram [2] supported with empirical OSS data.

Olbrich et al. [16] describe the study of two open-source systems over several years of development and focuse on two code smells in particular (the "God class" and "Shotgun Surgery" smells); different change behavior was observed for classes "infected" by code smells. More recently, Olbrich et al. [17] explore the change and fault proneness of "God" and "Brain" classes for systems ranging between seven and ten years old. Results showed that both smells were more fault and change prone, but when normalized for size were actually (and counter intuitively) less fault prone. Li and Shatnawi [18] investigated the link between bad smells and class error probability in an open-source system—some evidence to support high fault rates in smell code was reported. Van Emden and Moonen [19] investigated how the quality of code could be automatically assessed by checking for the presence of code smells and illustrated the feasibility of their approach through jCOSMO, a prototype code smell tool. A set of design flaws, including recognized code smells and a strategy, based on metrics for detecting those design flaws was described by Marinescu [20]. The mechanism was validated empirically. The same author refines that earlier work in [21]. Finally, Hamza et al. [22] provide an in-depth deconstruction of both Fowler's and Kerievsky's code smells [5] in an attempt to determine their overlap. While these studies have provided a basis for the area of code smells and the study of code smells, a range of open research issues persist. In the next three sections, we describe two empirical studies and one theoretical study (in that order) which question the viability of approaches to smell identification and eradication.

3. Refactorings Per Smell

3.1. Data Analysis. As a first part of our smell analysis, we describe the *potential* cost in time and effort of undertaking each of the 22 code smells. The basis of the analysis is that in Sections 4 and 5 we will see evidence of how limited smell eradication appears to be (Section 4) and some of the difficulties which arise when we attempt to sniff out code smells (Section 5). In this section, we provide a concrete suggestion as to why this might be the case. Earlier, we

stated how each of the code smells proposed by Fowler [1] could be eradicated by application of one or more other refactorings. Most refactorings (as well as having its own steps to completion) require other related refactorings to be undertaken in a nested relationship. Put another way, refactoring X might require refactoring Y, which in turn might require refactoring Z. Each of X, Y, and Z may also have other nested refactorings. All of X, Y, and Z can be extracted from Fowler's text as *remedies* for each of the smells. We can then posit that a factor inhibiting a developer addressing a code smell is the *total* number of refactorings that might need to be undertaken after following the "chains, induced by each of X, Y, and Z and used to remedy that smell. As part of our analysis, we therefore enumerated the refactorings that each of the smells induced, and this was achieved using a bespoke tool. Table 1 gives the 22 code smells listed in Fowler [1].

Figure 1 shows the potential number of refactorings that each of the 22 code smells requires. It is interesting that the smells observed in Section 4 are smells with relatively higher numbers of associated refactorings. The Large Class smell (number 10) has 40 associated refactorings. One of the reasons why this smell requires so many refactorings is due to requirement for the movement of methods to new class/classes and associated dependencies which, as we stated earlier, destroys class cohesiveness and forces the unpicking of all dependencies between methods. The Long Method smell (number 12) has 20 associated refactorings and the Lazy Class (number 11) 15 associated refactorings. As interestingly, these were the smells that we found difficult to tangibly identify from the ITWeb subsystem. On the other hand, the smells that we identified to be eradicated from the five OSS have relatively fewer required refactorings. Smell 1 actually has only 2 associated refactorings, and smell 16 has only 4 associated refactorings. Smells 7, 8, and 19 and 20 have relatively more associated refactorings overall but then again, we have no firm evidence that these were actually eradicated. Finally, the switch statement (smell 21) identified in one ITWeb class requires 16 separate refactorings in order to be eliminated—a relatively difficult smell to eradicate.

It would seem that developers might well eradicate smells, but they tend to be smells that require little effort when compared with others. Interestingly, and on a final note, in the developer survey carried out by Mäntylä [13], the Long Method smell stood out as the smell many developers "understood" the workings of most. In other words, conceptually speaking, developers know exactly what this smell arises from, the problems that it might pose and, more than likely, the means of eradicating this smell. One would therefore think that a greater understanding of a smell would imply that developers would naturally be more likely to want to eradicate that smell. However, we see that from Figure 1, the Long Method smell (smell 12) requires a relatively high amount of effort for its eradication. We therefore suggest that the effort required for the Long Method smell is a prohibiting factor for developers who might consider eradicating these smells. The lesson is that just because a smell is easy to understand does not mean it is easy to eradicate.

Table 1: Code smells and their description.

Smell	Description
(1) Alternative classes with different interfaces	Two classes appear different on the outside, but are similar on the inside
(2) Comments	Comments should describe why the code is there not what it does
(3) Data class	Classes should not contain just data, they should contain methods as well
(4) Data clumps	Data that belong together should be amalgamated rather than remain separated
(5) Divergent change	Changes to code should be kept local; too many diverse changes indicate poor structure
(6) Duplicated code	Eradicate duplicated code whenever possible
(7) Feature Envy	Class features that use other class" features should be moved those other classes
(8) Inappropriate intimacy	Classes should not associate with other classes excessively
(9) Incomplete library class	Avoid adding a method you need (and which does not exist in a library class) to a random class
(10) Large class	A class has too many methods
(11) Lazy class	A class is doing too little to justify its existence
(12) Long method	A method is too large; it should be decomposed
(13) Long parameter list	A method has too many parameters
(14) Message chains	Avoid long chains of message calls
(15) Middle man	If a class is delegating too much responsibility, should it exist?
(16) Parallel inheritance hierarchies	When you make a subclass of one class, you need to make a subclass of another
(17) Primitive obsession	Overuse of primitive types in a class
(18) Refused bequest	If inherited behavior is not being used, is inheritance necessary?
(19) Shotgun surgery	Avoid cascading changes; limit the number of classes that need to be changed
(20) Speculative generality	Code should not be added for "just in case" scenarios—it should solve current problems
(21) Switch statements	Polymorphism should be used instead of large switch statements
(22) Temporary field	Classes should not contain unnecessary fields

4. Open-Source Systems

4.1. Data Analysis. As a second part of our smell analysis, we use data from five, open-source Java systems as an empirical basis. These systems have been the basis of other empirical studies [2, 3, 23]. The criteria for initially choosing those

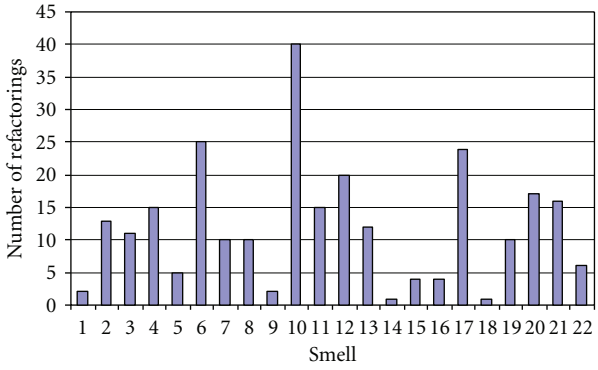

Figure 1: Code smells and their refactorings.

systems reported at length in those studies were that they had to be entirely Java systems and to have evolved over a minimum number of versions. From the same five systems we extracted a set of fourteen specific refactorings. The five systems studied were as follows.

(1) Antlr a framework for constructing compilers and translators using a source input of Java, C++ or C#. Antlr began with 153 classes and 31 interfaces. The latest version had 171 classes and 31 interfaces (five versions were studied).

(2) PDFBox a Java PDF library allowing access to components found in a PDF document. The initial system had 135 classes and 10 interfaces; the latest version of six had 294 classes and 52 interfaces.

(3) Velocity a template engine allowing web designers to access methods defined in Java. Velocity began with 224 classes and 44 interfaces. At the ninth version, it had 300 classes and 80 interfaces (nine versions were studied).

(4) Tyrant a graphical-based, fantasy adventure game, incorporates landscapes, dungeons and towns. The system began with 112 classes and 5 interfaces. At the ninth version, it had 138 classes and 6 interfaces (nine versions were studied).

(5) HSQLDB a relational database application supporting SQL. HSQLDB started with 52 classes and 1 interface. The latest version had 254 classes and 17 interfaces (four versions were studied).

The basis on which the initial study rests is that, to eradicate a smell, a specific set of refactorings need to be applied. Occurrences of fourteen specific refactorings were automatically extracted from multiple versions of these systems as part of an earlier study documented in [2]. The fourteen refactorings were chosen by two industrial developers as those most likely to be undertaken on a day-to-day basis and therefore ranged across OO concepts such as encapsulation and inheritance. Simpler refactorings for renaming/moving fields and methods were also included for the same reason. The refactorings were extracted by a bespoke tool and (in ascending order of frequency found in the five systems together with a brief description of each) are as follows.

(a) Encapsulate Downcast. "A method returns an object that needs to be downcast by its callers" [1]. (This was the least frequently applied refactoring.)

(b) Push Down Method. "Behavior on a superclass is relevant only for some of its subclasses" [1].

(c) Extract Subclass. A class has features used only in some instances—a subclass for that subset of features is created.

(d) Encapsulate Field. A field is made private.

(e) Hide Method. A method is made private.

(f) Pull Up Field. "Two subclasses have the same field. Move the field to the superclass" [1].

(g) Extract Superclass. Two classes have similar features. A superclass is created and common features moved.

(h) Remove Parameter. Parameter is unused by a method.

(i) Push Down Field. "A field is used only by some subclasses. Move the field to those subclasses" [1].

(j) Pull Up Method. Methods with identical results are moved to the superclass.

(k) Move Method. A method is moved to another class.

(l) Add Parameter. A parameter is added to a method.

(m) Move Field. A field is moved to another class.

(n) Rename Method. (This was the most frequently applied refactoring.)

4.2. Research Question. As part of the smell analysis, we first pose the question: *given the set of refactorings extracted from the five systems, which combination of those refactorings, applied to the versions of the five systems, have been used to remedy code smells*? In other words, from the data we collected on refactorings, do developers actually refactor (whether consciously or otherwise) to remedy smells and, if so, to what extent? The total set of 891 refactorings extracted by the tool over all versions of all systems was thus analyzed on a version-by-version basis to determine which smells they eradicated. The list of refactorings required for the reverse engineering process (i.e., to eradicate each smell) was provided in Fowler [1], and Table 1 gives the full list of 22 code smells. The process of deciding whether a smell had been entirely remedied required an exact match to be found between the list of refactorings specified by Fowler (to eradicate that smell) and a subset of refactorings extracted from the same version of a system. For a partial eradication, only a partial match between the extracted refactorings and that subset required to eradicate a smell was required. The smell analysis presented is based on *only* the refactorings extracted by the tool. The data on which refactorings had been extracted was analyzed using a spreadsheet, in which the frequency of each of the 15 refactorings was output on a version-by-version basis for each of the five systems. A sample of the set of 15 refactorings extracted was validated by the tool developers when the tool was run against the source code. This involved manual checking of the output

(i.e., the refactorings) against the Java source code. We thus have confidence in the correctness of the data and in the identification of the smells that we have identified as eradicated or partially eradicated.

Table 2 shows the five systems and the versions in each of the systems where some evidence of remedying of smells was found. For example, in versions 3 and 6 of the PDFBox system, a combination of refactorings was found to remedy smells 1 and 16. Column 3 of the table shows which smells were *entirely* remedied through application of refactorings. Column 4, on the other hand, shows the smells which *might* have been remedied.

Table 2 shows that we can only identify two smells as definitely having been remedied (i.e., smell 1 and 16). Smell 1 is "Alternative Classes with Different Interfaces." This smell occurs when two classes have a similar internal content but different external composition (i.e., in the parameter list). They should be amalgamated to present a common interface. Smell 16 is "Parallel Inheritance Hierarchies" where two separate inheritance hierarchies grow dependently and where creating subclasses in one requires subclasses to be created in the other. It is also worth noting that unexpectedly, later versions of the five systems did not show any greater propensity for smell eradication than earlier versions. This was surprising, since we might expect smells to worsen as a system evolved.

From Table 2, we see that following our analysis, only six of the remaining twenty smells *might* have been remedied according to column 4 (i.e., smells 3, 7, 8, 10, 19, and 20). Some of the fourteen refactorings identified from the systems have also been identified as "core" refactorings (i.e., are likely to be used frequently in multiple code modification scenarios). The "Move Method," "Move Field," and "Add Parameter" refactorings are typical examples [2]. These would be refactorings that we might expect a developer to apply as part of regular software maintenance and to be unconnected with conscious, intentional refactoring; it is difficult to know whether the developer actually set out to refactor or whether it was a byproduct of the day-to-day maintenance processes.

The results from Table 2 highlight the relative complexity of some smells over others, but the overriding message seems to be that only a small subset of smell eradications, from a small subset of the total number of versions from the five systems (13 versions from 33), were attempted. This claim has to be qualified with the caveat that we have no knowledge of whether any smell had been eradicated deliberately by the developer or whether the two refactorings were applied in combination at the same time to achieve the objective of smell eradication.

4.2.1. Smell Decomposition. Each of the six smells in Table 2 has a set of refactorings that need to be considered and then applied in order that they can be eradicated. For example, consider smell 1 "Alternative Classes with Different Interfaces." According to the Fowler mechanics, repeated application of the "Move Method" and "Rename Method" refactorings will remedy this smell. Equally, the "Parallel Inheritance Hierarchies" smell (smell 16) occurs when two separate inheritance hierarchies grow in a dependent fashion such that creating subclasses in one requires subclasses to be created in the other. Repeated application of the "Move Method" and "Move Field" refactorings would remedy this smell. It is the fact that these two smells require a relatively small number of frequently applied and overlapping refactorings (i.e., from the set of 14 extracted) that possibly account for the result in Table 2. All three refactorings (i.e., Move Method, Move Field, and Rename Method) appear in the top five refactorings from the list in Section 4.1. In other words, if the developer did set out to eradicate either of the two aforementioned smells, it may simply be because they only required the application of a set of two, relatively simple refactorings (and those that a developer regularly carried out) in each case.

Table 2: Smells eradicated (and partially eradicated).

System	Version	Smell remedied	"Partial" smells remedied
Antlr	2	1, 16	7, 8, 19, 20
PDFBox	3	1, 16	7, 8, 19, 20
	6	1, 16	3, 7, 8,19, 20
Velocity	2	1, 16	3, 7, 8, 19
	3	1, 16	3, 7, 8, 19
	5	1, 16	3, 7, 8, 19
	6	1, 16	7, 8, 19
	9	1	20
Tyrant	7	1	20
	8	1	3, 20
HSQLDB	1	1, 16	7, 8, 19, 20
	2	1, 16	7, 8, 10, 19, 20
	3	1, 16	7, 8, 10, 19, 20

Table 3 shows the total set of refactorings necessary to eradicate each of the six smells that were only partially remedied and illustrates that the frequently applied refactorings were, again, a common feature of the eradication process (i.e., Move Field, Move Method, and Rename Method appear often). The extent to which smells were only partially remedied is best illustrated with an example. From Table 3, code smell 10 (i.e., Large Class) requires the application of four refactorings in order to be eradicated. These four are "Extract Class," "Extract Subclass," "Extract Interface," and "Replace Data Value with Object". Only evidence of one of these four, namely, the "Extract Subclass" refactoring, was found from the extraction of refactorings by our tool. In other words, for this smell (and for many of the other five smells in Table 3) only a small minority of the *required* refactorings for eradication of smells were found to have been applied. The question arises as to why these smells were not totally eradicated? The "Large Class" smell (which occurs when a class is trying to do too much) requires the application of four refactorings; the first of these is "Extract Class" which decomposes a class into two or more separate classes. The "Extract Subclass" refactoring accounts

Table 3: Smells that were partially remedied.

Code smell	Name	Refactorings
3	Data Class	Move Method
		Encapsulate Field
		Encapsulate Collection
7	Feature Envy	Move Method
		Move Field
		Extract Method
8	Inappropriate Intimacy	Move Method
		Move Field
		Change Bidirectional Association to Unidirectional
		Replace Inheritance with Delegation
		Hide Delegate
10	Large Class	Extract Class
		Extract Subclass
		Extract Interface
		Replace Data Value with Object
19	Shotgun Surgery	Move Method
		Move Field
		Inline Class
20	Speculative Generality	Collapse Hierarchy
		Inline Class
		Remove Parameter
		Rename Method

for the case when the class being decomposed requires the creation of at least one subclass; the same rule applies to the "Extract Interface" refactoring in terms of decomposition of interfaces. The "Replace Data Value with Object" refactoring occurs when a data item needs additional data or behavior; the data item should be turned into an object. Each of these four refactorings required to eradicate the "Large Class" smell thus require significant effort to apply since they require either structural changes to the code or the introduction of objects to the system. While we did not collect all four refactorings, we suggest that the high-level structural nature of these refactorings was the reason why developers avoided their use in smell eradication.

The overriding message from the data in Tables 2 and 3 appears to be that while we can identify some evidence that refactorings associated with smells are undertaken, those refactorings appear to be ones that a developer might be liable to undertake anyway without any thought given to the eradication of any smell. For example, a developer might rename or move a method as part of general maintenance activity or in response to a fault fix. We are therefore skeptical about the claim that developers actively seek out code smells. The opposite may actually be the case; developers may actively avoid code smells because they are relatively difficult to tackle.

4.2.2. The Remaining Smells. Smell 3 "Data Class" is a class that has fields and getting and setting methods for those fields and are therefore merely data holders for other classes. This implies that they only exist to be manipulated by other classes. Smell 7 is "Feature Envy," which occurs when the methods of one class use the methods of another class excessively. Smell 8, "Inappropriate Intimacy," arises when two classes are coupled excessively to each other. The "Shotgun Surgery" smell arises when a change in one class requires cascading changes in several other classes. "Speculative Generality" arises when a developer adds functionality in the belief that it will be needed later on. We note that Mäntylä's taxonomy [13] rates this smell alongside duplicate code and dead code in terms of the harmful effect it might have on a system. While all of these smells require at least one simple refactoring, they also require the application of at least one "complex" refactoring. We suggest that the complexity of eradicating a smell (in some cases) is a factor in developers avoiding smell eradication.

5. A C# Proprietary System

5.1. An Aggressively Refactored System. The final part of our analysis is exploration of a C# subsystem for a web-based, loans system providing quotes and financial information for online customers; henceforward and for system anonymity we will refer to this system as simply "ITWeb." We examined two versions of one of its subsystems. The first, an early version, comprised 401 classes. A later version (henceforward version n) had been the subject of a significant refactoring effort to amalgamate, minimize as well as optimize classes—we were given no information as to which version it represented; it comprised 101 classes only and had thus been reduced in size by 300 classes through a process of aggressive refactoring (through merging of classes and optimization of others). For the purposes of this second analysis, we focused on four smells which, arguably, should be easily identifiable from the source code via simple metrics. These were as follows.

(1) Large Class. A class is trying to do too much, identified by a relatively large number of methods. Such a class is difficult to maintain and thus should be refactored—the class should be decomposed into two or more classes.

(2) Long Method. A method is doing too much, identified by its large number of executable statements. In the same way that as that for Large Class. The method should be decomposed into two or more methods.

(3) Lazy Class. A class is not doing enough to justify its existence, identified by a small number of methods and/or executable statements; it should be merged with its nearest, related class.

(4) Comments. There are conflicting opinions on the role that comments play in code from a smell perspective. Large numbers of comments in theory are useful, but, on the other hand might suggest that the relevant code is overly complex and thus needs significant explanation; an alternative viewpoint is that the code has been modified significantly and

the comments reflect the activity around a method or methods of a class over the course of its lifetime. Excessive comments should be treated with care—they may be a smell indicating problematic code.

The SourceMonitor tool [24] was used to extract a set of smell-relevant metrics [25] from each version.

Metric 1 (Average Methods per Class). It is defined as the average methods per class for all class, interface, and struct methods. It is computed by dividing the total number of methods by the total number of classes, interfaces, and structs.

Metric 2 (Average Statements per Method). The total number of statements found inside a class divided by the number of methods.

Metric 3 (Average Calls per Method). The average number of calls to other methods inside all methods in a class (i.e., intracoupling). This metric does not include calls to methods of other classes in the same way that, for example, the Coupling between Objects metric of Chidamber and Kemerer does [26].

Metric 4 (Average Class Complexity). It is defined as the sum of the complexities of the methods in a class divided by the number of methods and is in line with the definition provided by McConnell [27]. The complexity of a method is the count of the number of unique paths through a method. Each method therefore has a minimum complexity of "1" if it comprises just one path. A value of "1" is added to the value of the metric for each branch statement (if, else, for, while, etc.; a "1" is also added for each "||" or "&" operator in an "if" or "while" statement).

Metric 5 (Percentage Comments). This metric reflects the percentage of lines of code accounted for by comments.

For each of these four metrics individual class values were also collected (i.e., the methods, statements, calls per method, complexity, and comments per class). We note that in several cases, a file can contain more than one class, in which case the average reported is that for the set of classes rather than the individual class.

5.2. Research Question. The research question on which we analyze ITWeb is as follows. *Can we expect the four aforementioned code smells to occur frequently in a system when we deliberately set out to "sniff" them*? Moreover, we might expect that since the system was aggressively refactored from version 1 to *n*, we might find less occurrences of the set of smells in the later version than in the earlier. Table 4 shows the summary data for versions 1 and *n* for the ITWeb system. We see that the average number of methods, average size (statements) of methods, average calls per method, average complexity decrease from version 1 to version *n* suggesting that the extensive refactoring that occurred between version 1 and *n* succeeded in reducing both class size and complexity. Percentage of comments also decreased from version 1 to version *n*, suggesting that developers deemed the removal of comments as a necessary aspect of the refactoring process. The question that arises is whether and/or to what extent either version presented opportunities for smell eradication? We explore each of the four listed smells in the order described to find out.

5.2.1. Large Class. To identify occurrences of the Large Class (LC) smell in version 1, one way of achieving this would be to order classes on descending Average Methods per Class and refine the search from there. By doing this, we find that the class with the largest number of methods is a *sealed* C# class (i.e., it cannot be inherited) called `PageController.cs`. This is an architectural, pattern-based class, essential for the coordination of ITWeb, and contains 80 methods. (Fowler defines the page controller pattern as "An object that handles a request for a specific page or action on a website".) Inspection of the code for this class revealed that each method handled one of a number of functionally cohesive requests for web page details. For example, there were methods to `SaveAccountDetails`, `SaveApplicationDetails`, and `SaveBrokerContactDetails`, and so forth. The average complexity of this class was 2.08, well below the average complexity for version 1. The number of calls per method was 4.5, well above the average. This last metric presents a conflict: a strong interdependence and coupling between the methods in this class is generally considered to contribute positively to the cohesiveness of the class, but equally would pose a huge problem if we wanted to decompose the class. The classes with the second and third largest numbers of methods are Controller and Navigation-based classes (`LoanSessionController.cs` and `PageNavigator.cs`, resp.), again with similar functionality to `PageController.cs`.

Ordering version *n* in the same way, we find that the maximum number of methods is 12 and belongs to a class called `ResultRowDTO.cs`. This is another architectural, pattern-based Data Transfer Object (DTO) [28]. A DTO wraps up data for transfer between two processes, possibly over a network, to prevent the overhead of multiple (remote) calls. Inspection of the code for this class confirmed that each of the 12 methods contained only a single "get" and "set" method. The average complexity of this class was 1, well below the average complexity for the subsystem of 1.13, and the number of calls per method was zero. In contrast to being classified as a smell, this class is a key element of the system architecture and has desirable properties only. The median of class size was 3 for this version, reinforcing the difficult question as to "which class to choose for eradication?" The problems that arise with the LC smell are therefore (a) deciding on what exactly constitutes a "large" class (a largely arbitrary choice) and (b) the fact that coupling between methods adds to class cohesion [29], yet makes an LC smell eradication more problematic due to dependencies. Figures 2(a) and 2(b) show the distribution of average methods per class in version 1 (Figure 2(a)) and version *n* (Figure 2(b)), respectively. The scale on the *y*-axis is an indication of the extent to which the average number of methods per class was reduced as a direct result of the refactoring effort. Only one

Table 4: Summary Data for ITWeb.

Version	Classes	Ave. no. methods	Ave. statements/method	Ave. calls/method	Ave. complexity	%Comments
1	401	5.59	4.03	2.24	1.70	4.0
N	101	3.35	1.79	1.29	1.13	0.3

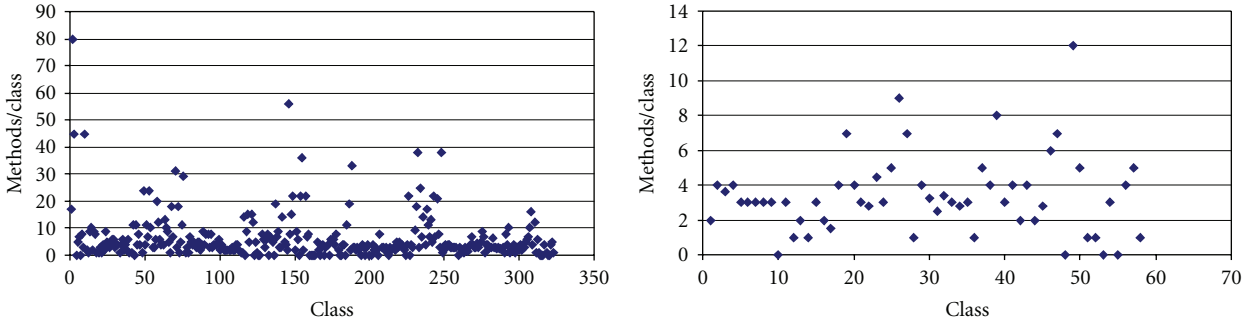

Figure 2: (a) Methods/class (version 1). (b) Methods/class (version n).

class in version n has an average number of methods above 10. This compares with 45 values exceeding that value in version 1.

We conducted an 80/20 analysis of the values in each of Figures 2(a) and 2(b) to determine the distribution changes that had taken place from version 1 to version n [30]. For Figure 2(a), 80% of values were found in approximately 136 (33%) of the 401 classes in total. For Figure 2(b), 80% of values were found in approximately 55 (approximately 54%) of the 101 classes, emphasizing the positive change in distribution of the values in the later version and a more even distribution as a result of the refactoring process. A short "head" and "long" tail are therefore less evident in the later version than in the former. The standard deviation of the values in Figure 2(a) is 8.61, compared with just 2.26 for Figure 2(b).

We note that the scale on the x-axis of Figures 2(a) and 2(b) represent files, within which more than one class may be housed (this explains the value of 350 and 70 on the respective axes and does not reflect the number of classes).

5.2.2. Long Method. If we now turn to the Long Method (LM) smell, one way of identifying such a smell is to order the set of classes in each version on Statements per Method and then refine the search from there. If we order version 1 in this way, we find that the class `ComparisonEngine.cs` contains the method with the highest number of statements. Inspection of the code revealed this method to contain one large *switch* statement comprising 340 statements. The switch statement is actually a smell itself (see Table 1, number 21), since OO suggests the use of the more appropriate OO facet of polymorphism instead of large decision control constructs [1]. However, we see no value in decomposing this method since for a web application switch statements effectively represent user interface choice (an essential aspect for guiding the online process).

Doing the same for version n, we found that the largest method was again a DTO class called `LoanDTO.cs` containing just one method. Inspection of the code revealed the method to be a set of similar, executable statements for returning a code to the main program depending on the values of user-filled fields (e.g., combinations of company name seeking online quotes, company logo, payment details, and payment rates). The average complexity of this class was 5, well above the average (1.13). However, the very nature of each method (comprising multiple "if" statements) does contribute to complexity as we have defined it. That said however, we see no obvious benefits to simplifying code complexity where the purpose of the code is obvious and essential to the workings of the system.

Three overriding messages from the LM analysis emerge. First, by necessity, some methods (often the longest) exist for a good reason. Second, searching out one smell can often lead to the identification of other, potentially more harmful smells. One aspect of the analysis presented is that by avoiding some smells, we may inadvertently miss others. Moreover, we have to consider the possibility that smells are created due to eradication of others; in the same vein, a study by Pietrzak and Walter used knowledge of already detected smells and the interrelationships between smells to detect further smell types [31]. Finally, necessary complexity and how that manifests in code is often infeasible to remove when existing code communicates the purpose of the code.

Figure 3(a) shows the distribution of statements per method for version 1 of the ITWeb system; Figure 3(b) shows the same distribution for version n of the same system. As per the values in Figures 2(a) and 2(b), the contrast between the two set of values is marked in terms of the reduction shown in the second of the figures. The maximum value in Figure 3(b) is 8.33 compared with 38.63 in version 1 (Figure 3(a)). An 80/20 analysis of the values in each of Figures 3(a) and 3(b) revealed that for Figure 3(a), 80% of values were found in approximately 113 (approximately

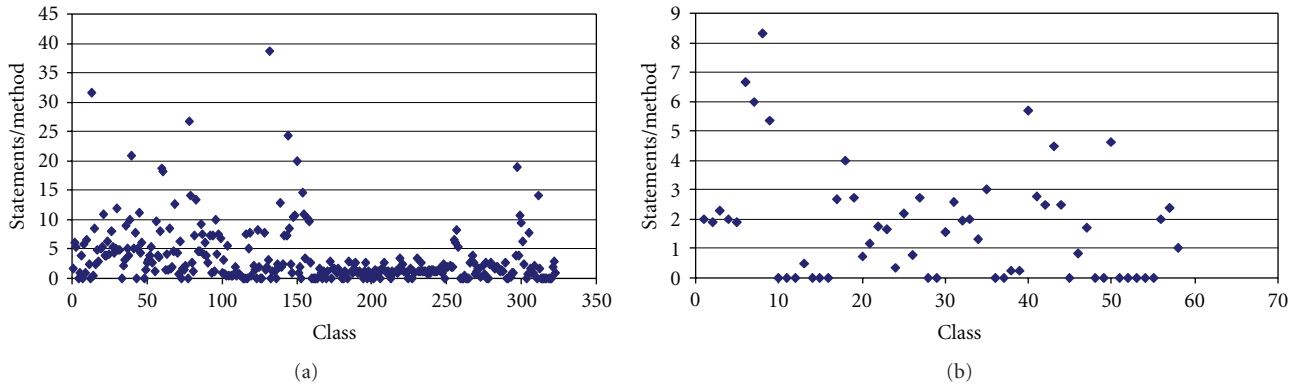

Figure 3: (a) Statements/method (version 1). (b) Statements/method (version n).

```
public enum Fulfilment Type
{
None= 9, Site= 11, Ref= 12, BkrRef= 13
}
```

Figure 4: Enumerated Type.

28%) of the 401 classes in total. For Figure 3(b), 80% of values were found in 23 (approximately 22%) of the 101 classes, emphasizing the change in distribution of the values in the later version and the need for fewer statements per method in the system as a whole. This is supported through the standard deviation with values in Figure 3(a) of 4.82, compared with 1.92 for Figure 3(b).

5.2.3. Lazy Class. If we now turn to the Lazy Class smell, one route to its identification would be to order classes on statements (ascending) and work downwards thereafter. If we order version 1 in this way, we find that 16 classes had just a single statement. Many of these classes were single, type-based classes similar to that shown in Figure 4. Many other small classes with 3 statements were collection-based classes (38 classes fell into this category) which returned values from a collection, based on a parameter index value passed. In other words, each of these had a specific and cohesively tight functionality and would not, at first impressions, be candidates for amalgamation into other classes.

The complexity of such classes was also found to be slightly lower than average (typically 1.67 compared with 1.70 overall). Again as per the Large Class smell, we note a conflict between the need for a class to remain cohesive (and hence reasonably small) and the search for lazy (small) classes. Put another way, detecting lazy classes may be flawed by definition. For version n, the class with smallest number of statements was `ProductDetailsModel.cs`, comprising 2 statements; this is an empty class. The next smallest is class `LoansServiceAction.cs` which contains 3 statements comprising an enumerated data type only. Interestingly, 3 of the 10 smallest classes are "view" interfaces and not classes.

5.2.4. Comments. One strategy for identifying classes with large numbers of comments would be to list all classes on ascending percentage of comment lines and examine the code in the resultant classes. By doing this for version 1, we find that four classes all comprise exactly 69% comments. Inspection of the code revealed that none of these classes contained any methods or any other features associated with the OO paradigm. Each of these classes in fact revealed no executable C# code. The files contained information on assembling .net version information and standards for creating new versions of code. The files contained mostly comment lines surrounding that assembly information. The class with the next highest number of comment lines was a class with 55% of comments and represented the initial page requirements for the ITWeb application. For example, the questions that a user had to answer before receiving a quote were an integral part of this class. Intenestingly, inspection of the code in the class revealed old code to have been commented out rather than deleted. In other words, the high percentage of comments was not attributed to comments in the true spirit of the word, but to code that was no longer needed. There were only seven "live" methods in the class, with very few comments associated with these methods.

For version n, the maximum percentage of comments amongst the set of 101 classes was just 5.1%, and this belonged to a class called `ServiceErrorHandler.cs`. Inspection of the code for this class showed a very sparse distribution of comments, as might be expected from the percentage of comments it contains and the aggressive nature of the refactoring for the system as a whole. Only 10 of the 101 classes had a percentage number of comments above zero, and seven of those ten classes contained less than 1% comments. It appears that the process of refactoring (from version 1 to n) caused a dramatic reduction in the level of comments in the subsystem. All "commented out" lines of code had also disappeared from the system. It is also interesting that 91 classes in version n had zero comments. Discussions with the project leader suggest that the refactored subsystem had become a stable part of the overall system, experienced very few faults, and thus required little maintenance. Figure 5(a) shows the percentage comments for the classes in version one and in Figure 5(b) the same distribution for version n of the same system. The contrast between the range of comments in version 1 and

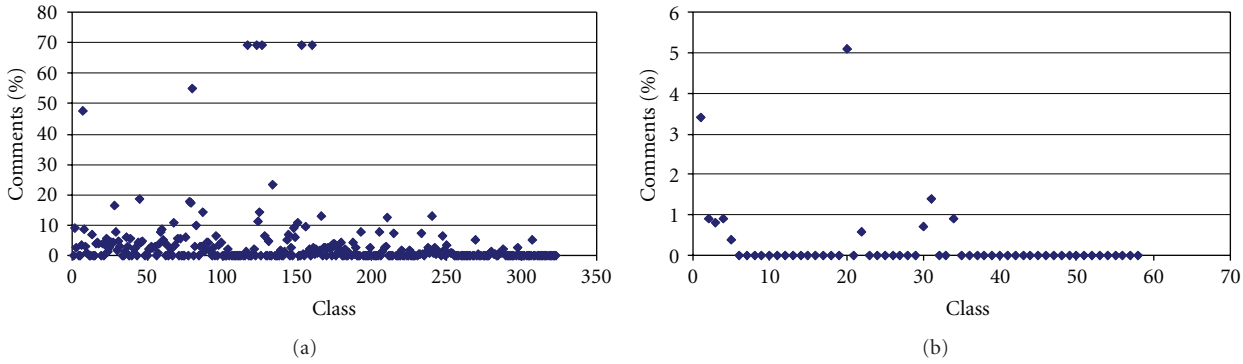

Figure 5: (a) Percentage comments (version 1). (b) Percentage comments (version *n*).

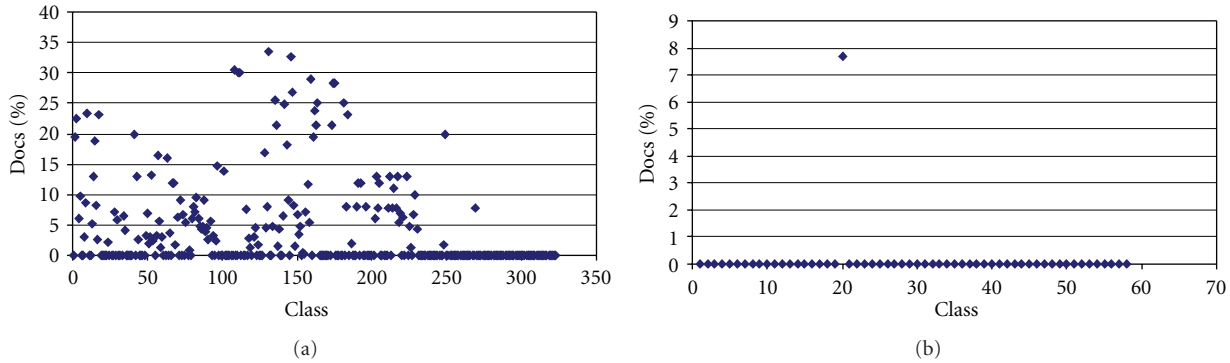

Figure 6: (a) Percentage Docs (version 1). (b) Percentage Docs (version *n*).

version *n* is clear from the two figures in terms of the low percentages in Figure 5(b).

We conducted an 80/20 analysis of the values in each of Figures 5(a) and 5(b) to determine the distribution changes that had taken place from version 1 to *n*. For Figure 5(a), 80% of values were found in 56 (approximately 14%) of the 401 classes in total. For Figure 5(b), 80% of values were found in 6 (approximately 10.34%) of the 101 classes. A marginally shorter head and a longer tail are therefore evident in Figure 5(b). However, that said, the standard deviation of the values in Figure 5(a) is 9.76, compared with just 0.83 for Figure 5(b). This result for comments illustrates that as a result of aggressive refactoring, fewer comments were needed in the refactored version of the system; the standard deviation values suggest that the variation in numbers of comment lines in classes was also less as a result. In other words, one side-effect of refactoring might have been less complex code and the need for fewer comments in fewer classes as a result.

It would appear that, of the four smells that we have considered, reduction in number of comments may be a clear "byproduct of aggressive refactoring effort. It would also seem that addition of comments (based on the high number of zero-values from Figure 5(b)) was not a consideration for the refactored subsystem. We also have to consider the "doc" comments in each version (these are automatically created through the inclusion of XML tags in the source code). The compiler will search for all XML tags in the source code and create an XML documentation file accordingly. Figure 6(a) shows the percentage of "Doc" comment lines in the classes of version 1 and Figure 6(b) that for version *n*. To complete our analysis, we analysed the occurrence of "Doc" comments on an 80/20 basis for Figures 6(a) and 6(b). For version 1, 66 of classes comprised 80% of the percentage Docs (approximately 16%) of the 401 classes. For version *n*, 1 single class contained 100% of all Docs as can be seen from Figure 6(b). All remaining 100 classes comprised zero Docs. The result reflects the nature of the refactoring effort invested and the reduction in both percentage comments (Figures 5(a) and 5(b)) and Docs that the aggressive refactoring policy induced.

6. Threats to Validity

Several criticisms or threats could be leveled at the study. First, the analysis presented makes the assumption that a developer has no sense of the effort required to eradicate a smell, in other words, that a developer is oblivious to the presence of smells, and their eradication. This stance might be naïve on our part. A developer might be able to detect a smell; the same developer might also be well aware of the relative advantages of leaving that smell to become a "stench" or to eradicating that smell. Developers also have to make difficult choices as to how they allocate their time. We have no evidence that developers actively avoid smells. Second, there may be many other types and flavors of code smell that

a developer would consider eradicating before those listed in Table 1. We cannot assume that the 22 smells listed in Fowler [1] are necessarily the definitive set of smells. That said, we see this work as a worthwhile start to establishing the parameters through which code smells on a longitudinal basis [32] can be properly and fully explored. Third, we cannot discount the extent to which in the empirical studies presented tools were used to identify the source of code smells. In the case of the ITWeb subsystem, we know of no tools used in the process of smell eradication. Future work will focus on exploring two issues: we intend exploring the test implications of smells and second, on validating the practical and theoretical results with industrial developers and on a range of open-source systems [33]. Finally, as an emerging and developing area, we see potential for a systematic code smell literature review to assess the state-of-the-art and establish key research themes.

7. Conclusions/Future Work

In this paper, we have described three studies based on code smells, two of an empirical nature and one a theoretical analysis which attempts to place those studies in context. First, we provided a theoretical suggestion as to why eradicating smells might be problematic. The second study tried to establish whether developers eradicated smells based on the refactorings applied to five open-source systems. The final study was of a proprietary C# system where aggressive refactoring had taken place and a large reduction in system size had been experienced. The key findings of the study presented are as follows; first, no evidence of widespread eradication of smells was found; evidence of simple refactorings in favor of more "complex" refactorings was found. Developers might either be unaware that they are actually eradicating smells or simply avoid complex smells (if empirical data extracted from systems is used as a guide). Second, a wide range of conflicts, contradictions, and anomalies soon emerge when we first try to identify code that smells. This makes the task of identifying "real" smells problematic and possibly prohibitive. Finally, the projected effort in terms of related refactorings that need to be undertaken to eradicate a smell may also be a prohibiting factor. We urge more studies on smells, and to assist in a small way, the data from studies 1 and 3 can be made available upon request of the lead author.

Acknowledgments

The authors thank the anonymous reviewers for their comments which have contributed significantly to improvements in the paper. The research in this paper was supported by a grant from the UK Engineering and Physical Sciences Research Council (EPSRC) (Grant no.: EP/G031126/1).

References

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

[2] S. Counsell, Y. Hassoun, G. Loizou, and R. Najjar, "Common refactorings, a dependency graph and some code smells: an empirical study of java OSS," in *Proceedings of the 5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE '06)*, pp. 288–296, Rio de Janeiro, Brazil, September 2006.

[3] S. Counsell, "Is the need to follow chains a possible deterrent to certain refactorings and an inducement to others?" in *Proceedings of the 2nd International Conference on Research Challenges in Information Science (RCIS '08)*, pp. 111–122, Marrakech, Morocco, June 2008.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '00)*, pp. 166–177, Minneapolis, Minn, USA, October 2000.

[5] J. Kerievsky, *Refactoring to Patterns*, Addison-Wesley, 2005.

[6] T. Mens and A. van Deursen, "Refactoring: emerging trends and open problems," in *Proceedings 1st International Workshop on REFactoring: Achievements, Challenges, and Effects (REFACE '03)*, Univ. of Waterloo, 2003.

[7] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[8] R. Najjar, S. Counsell, G. Loizou, and K. Mannock, "The role of constructors in the context of refactoring object-oriented software," in *IEEE European Conference on Software Maintenance and Reengineering*, pp. 111–120, Benevento, Italy, March 2003.

[9] W. Opdyke, *Refactoring object-oriented frameworks*, Ph.D. thesis, Univ. of Illinois, 1992.

[10] M. V. Mäntylä, J. Vanhanen, and C. Lassenius, "Bad smells—humans as code critics," in *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pp. 399–408, Chicago, Ill, USA, September 2004.

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

[12] M. V. Mäntylä and C. Lassenius, "Drivers for software refactoring decisions," in *Proceedings of the 5th ACM-IEEE International Symposium on Empirical Software Engineering (ISCE '06)*, pp. 297–306, Rio de Janeiro, Brazil, September 2006.

[13] M. Mäntylä, *Bad smells in software–a taxonomy and an empirical study*, M.S. thesis, Helsinki University of Technology, Software Business and Engineering Institute, 2003.

[14] F. Khomh, S. Vaucher, Y.-G. Guéehéeneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proceedings 9th International Conference on Quality Software (QSIC '09)*, C. Byoung-ju, Ed., pp. 305–314, August 2009.

[15] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings 16th Working Conference on Reverse Engineering (WCRE '09)*, pp. 75–84, Antwerp, Belgium, October 2009.

[16] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: a case study of two open source systems," in *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09)*, pp. 390–400, Lake Buena Vista, Fla, USA, October 2009.

[17] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems," in *Proceedings of IEEE International Conference on Software Maintenance (ICSM '10)*, Timisoara, Romania, September 2010.

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

[19] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE '02)*, Richmond, Va, USA, 2002.

[20] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pp. 350–359, Chicago, Ill, USA, September 2004.

[21] R. Marinescu, "Measurement and quality in object-oriented design," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*, pp. 701–704, September 2005.

[22] H. Hamza, S. Counsell, G. Loizou, and T. Hall, "Code smell eradication and associated refactoring," in *Proceedings of the European Computing Conference (ECC '08)*, Malta, September 2008.

[23] S. Counsell and S. Swift, "Refactoring steps, java refactorings and empirical evidence," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference*, pp. 176–179, Turku, Finland, August 2008.

[24] SourceMonitor Tool, http://www.campwoodsw.com/sourcemonitor.html.

[25] N. Fenton and S. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, International Thomson Press, London, UK, 2002.

[26] S. R. Chidamber and C. F. Kemerer, "Metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[27] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, 2004.

[28] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison Wesley, 2003.

[29] S. Counsell, S. Swift, and J. Crampton, "The interpretation and utility of three cohesion metrics for object-oriented design," *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 2, pp. 123–149, 2006.

[30] R. Wheeldon and S. Counsell, "Power law distributions in class relationships," in *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 45–54, Amsterdam, The Netherlands, 2003.

[31] B. Pietrzak and B. Walter, "Leveraging code smell detection with inter-smell relations," in *Proceedings of the 7th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP '06)*, vol. 4044 of *Lecture Notes in Computer Science*, pp. 75–84, Oulu, Finland, June 2006.

[32] C. F. Kemerer and S. Slaughter, "An empirical approach to studying software evolution," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 493–509, 1999.

[33] T. T. Dinh-Trong and J. M. Bieman, "The FreeBSB project: a replication case study of open source development," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 481–494, 2005.