Exploring the Impact of Inter-smell Relations on Software Maintainability: An Empirical Study

Aiko Yamashita
Simula Research Laboratory
Lysaker, Norway
aiko@simula.no

Leon Moonen
Simula Research Laboratory
Lysaker, Norway
leon.moonen@computer.org

***Abstract*—Code smells are indicators of issues with source code quality that may hinder evolution. While previous studies mainly focused on the effects of individual code smells on maintainability, we conjecture that not only the individual code smells but also the *interactions* between code smells affect maintenance. We empirically investigate the interactions amongst 12 code smells and analyze how those interactions relate to maintenance problems. Professional developers were hired for a period of four weeks to implement change requests on four medium-sized Java systems with known smells. On a daily basis, we recorded *what specific problems* they faced and *which artifacts* were associated with them. Code smells were automatically detected in the pre-maintenance versions of the systems and analyzed using Principal Component Analysis (PCA) to identify patterns of *co-located code smells*. Analysis of these factors with the observed maintenance problems revealed how smells that were co-located in the same artifact interacted with each other, and affected maintainability. Moreover, we found that code smell interactions occurred across *coupled* artifacts, with comparable negative effects as same-artifact co-location. We argue that future studies into the effects of code smells on maintainability should integrate dependency analysis in their process so that they can obtain a more complete understanding by including such *coupled interactions*.**

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

I. Introduction

The presence of code smells indicates that there are issues with code quality, such as understandability and changeability. This can lead to a variety of maintenance problems, including the introduction of faults [1]. Beck and Fowler described 22 code smells and associated each of them with refactoring strategies that can be applied to counteract the potentially negative consequences of having these smells present in your source code [1]. However, code smells are only *indicators* of potentially problematic code. Not all smells are equally harmful, and some of them may not even be harmful in particular contexts. In addition, applying refactoring strategies implies a certain cost and risk, e.g., any changes made to the code may introduce unwanted side effects and could trigger faults in the system. Therefore, we need to better understand the relationship between code smells and maintenance problems.

When analyzing the relation between individual code smells and maintainability [2], we observed that several code smells tended to be present in the same artifact. Based on these observations, we conjectured that interaction effects between *co-located smells* (i.e., smells located in the same artifact) can *intensify problems* caused by individual code smells or lead to *additional, unforeseen maintenance issues*. The notion of inter-smell relations has only been brought to attention recently, but the concept seems promising for understanding the nature and effects of code smells. Inter-smell relations were introduced by Pietrzak and Walter [3] to support more accurate detection of code smells. An example inter-smell relation they propose is *plain-support*, which is defined as follows: "*...smell B is supported by smell A if the existence of A implies with sufficiently high certainty the existence of B. B is then a companion smell of A, and the program entities (classes, methods, expressions etc.) burdened with A also suffer from B...*" While Pietrzak et al. introduced inter-smell relations in the context of code smell detection, we argue that this notion can also help to better understand how code smell *interaction effects* can cause problems for developers.

This paper reports on an empirical study that investigates interactions among twelve different code smells and how these interactions can lead to maintenance problems. It is based on an industrial case study in which six professional software engineers were hired to maintain four medium-sized Java systems with equivalent functionality but dissimilar designs for a period of up to four weeks. During that time, they were asked to implement a number of change requests. Via interviews and think-aloud sessions, we recorded on a daily basis detailed reports about the maintenance problems that were faced, and which artifacts were associated with them. Afterwards, we used tools to automatically detect the code smells that were present in the original "pre-maintenance" versions of the systems and applied Principal Component Analysis (PCA) to find out how code smells were co-located (i.e., find clusters of smells present in the same artifact). The reported problems were analyzed in detail to investigate how co-located smells interacted with each other, and how this interaction related to the problems experienced during maintenance. In addition, we found that interaction occurred between code smells distributed across *coupled artifacts* ("coupled smells" hereafter) and analyzed how this affected maintenance.

The remainder of this paper is structured as follows: Section 2 presents the background and related work. Section 3 describes the study design, the systems under analysis and the maintenance tasks. Section 4 presents and discusses the results. Section 5 concludes and presents future work.

978-1-4673-3076-3/13/$31.00 © 2013 IEEE

II. Theoretical Background and Related Work

A code smell is a suboptimal design choice that can degrade different aspects of code quality, such as understandability and changeability, and may also lead to the introduction of faults [1]. Beck and Fowler [1] informally described 22 code smells and associated them with refactoring strategies to improve the design. In the last decade, code smells have become an established concept for patterns or aspects of software design that may cause problems for further development and maintenance of these systems [4]. Code smell analysis allows people to integrate both assessment and improvement into the software evolution process itself.

Van Emden and Moonen [5] provided the first formalization of code smell detection and developed an automated code smell detection tool for Java. Mäntylä [6] and Wake [7] proposed two initial taxonomies for code smells. Mäntylä investigated how developers identify and interpret code smells and how these methods compare to results from automatic detection tools. Examples of recent approaches for code smell detection can be found in [8–17]. Automated detection has been implemented in commercial tools, such as Borland Together[1] and InCode[2].

Previous studies have empirically investigated the effects of individual code smells on different aspects related with maintainability, such as *defects* [18–22], *effort* [23–26] and *changes* [15, 27, 28]. From these earlier empirical studies, only the study by Abbes et al. [26] has brought up the notion of interactions between code smells. They conducted an experiment in which 24 students and professionals were asked questions about the code in six Open Source Systems (OSS). The authors concluded that classes and methods identified as God Classes and God Methods had no effect on effort or quality of responses in isolation, but when they appeared together, they led to a statistically significant increase in response effort and a statistically significant decrease in the percentage of correct answers. We believe that the explanatory and predictive power of code smells can be improved by considering and investigating inter-related code smells, rather than only focusing on the study of individual code smells. This study attempted first to identify patterns of co-located code smells via PCA and subsequently explored the effects of interactions between code smells on the incidence of maintenance problems.

III. The Empirical Study

A. Systems Under Analysis

In 2003, Simula Research Laboratory's Software Engineering department sent out a tender for the development of a web-based information system to keep track of their empirical studies and resulting scientific publications. Based on the submitted bids, four Norwegian consulting companies were hired to independently develop a version of the system using the same requirements and specifications. More details on the original development projects can be found in [29]. The four development projects led to the creation of four systems with the same functionality. We will refer to them as System A, System B, System C, and System D. The systems were primarily developed in Java, and they all have similar three-layered architectures. Although the systems exhibit nearly identical functionality, there were substantial differences in how they were designed and coded. An overview of their size in lines of code (LOC) and number of Java files is shown in Table I.

TABLE I
Size of the Systems Analyzed (LOC)

System	A	B	C	D
LOC	7937	14549	7208	8293
No. Java files	63	167	29	118

The systems were all deployed over Simula Research Laboratory's Content Management System (CMS), which at that time was based on PHP and a relational database system. The systems had to connect to the database in the CMS to access data related to the research at Simula Research Laboratory as well as information on the publications.

B. The Maintenance Tasks and the Developers

In 2008, Simula Research Laboratory introduced a new CMS called *Plone*[3], which interfered with the operation of the systems. Consequently, it was necessary to modify the systems so they could operate in the new environment (i.e., an adaptive maintenance task was required). In addition, it was required to extend the systems with additional functionality. Two Eastern European software companies were hired to do the maintenance tasks between September and December 2008 at a total cost of 50,000 Euros. The maintenance tasks are briefly described in Table II and were completed by six different developers. The developers were recruited from a pool of 65 participants of a previously completed study on programming skill [30]. Additional information about the skill scores used for this purpose can be found in [30]. The developers were evaluated to ensure they had sufficient English skills for the purpose of our study, and they were explicitly asked if they were willing to take part in the planned study. As part of the study, all developers individually completed each of the three tasks (in the order specified in Table II) on two different systems. These developers first performed all tasks on one system and then repeated the tasks on the second system. The systems were randomly assigned to developers with controls in place for equal representation. Clearly, there is a learning effect from repeating the same tasks on a second system. However, this effect is not considered a threat to the validity of this particular study, since (i) it reflects realistic situations where developers have relevant experience for performing similar tasks, (ii) the speed with which developers perform their tasks

[1] http://www.borland.com/us/products/together

[2] http://www.intooitus.com/products/incode

[3] http://plone.org

TABLE II
MAINTENANCE TASKS

Task	Description
1. Adapting the system to the new Simula CMS	In the past, systems had to retrieve information through a direct connection to a relational database within Simula's domain (mainly information concerning the researchers at Simula and publications associated or derived from the different studies). Currently, Simula uses a CMS based on the Plone platform, which uses an Object Oriented database called ZODB[51]. In addition, the Simula CMS database previously contained unique identifiers for employees and publications that were based on Integer type. Now, a Char type is used as a unique identifier for both employees and publications. Task 1 consisted of modifying the data retrieval procedure by consuming a set of web services provided by the new Simula CMS to access all the information associated with employees and publications.
2. Authenticating via web services	Under the previous CMS, authentication was done through a connection to a remote database, and this process used the authentication mechanisms available at that time for the Simula website. Task 2 consisted of replacing the existing authentication procedures by calling a web service for this purpose.
3. Adding new reporting functionality	A new functionality consisted of introducing a set of options for configuring personalized reports. In these reports, the user should be able to choose the type of information related to a study to be included in the report, set as inclusion criteria a list of the people responsible for the study, sort the resulting studies according to when the study was finalized, and group the results according to the type of study. The configuration of the report must be stored in the systems' database and should be editable only by the owner of the report configuration.

is of no importance, and (iii) repeating the same tasks actually helps developers to contrast maintainability across the systems.

C. Study Design

1) The Process: First, the developers were given an overview of the tasks (e.g., the motivation for the maintenance tasks and the expected activities). Then they were provided with the specification of the maintenance tasks. When needed, they could discuss the maintenance tasks with a researcher; one was always present at the site during the entire duration of the project. We had daily meetings with the developers where we tracked their progress and the problems they encountered.

Think-aloud sessions were conducted every other day at a random point during the day, and they lasted for 30 minutes. Acceptance tests and individual open interviews, which had a duration of 20-30 minutes, were conducted once all three tasks were completed. In the open-ended interviews, the developers were asked about their opinions of the system, e.g., about their experiences when maintaining it.

Eclipse was used as a development tool along with MySQL[4] and Apache Tomcat[5]. Defects were registered in Trac[6], and Subversion[7] was used as the versioning system. A plug-in for Eclipse called Mimec [31] was installed on each developer's computer to log all the user actions performed at the graphical user interface (GUI) level with millisecond precision.

2) Code Smells Analyzed: Twelve code smells were automatically identified in the original *"pre-maintenance" versions* of the systems, using Borland Together and InCode. Table III presents descriptions of the code smells that were detected (based on [1, 32]). The detection strategies used in the tools are based on [33] (see the Appendix in our technical report [34] for more details on the detection parameters). Even though it is not part of the 22 smells defined by Fowler and Beck, the design principle violation called Interface Segregation Principle Violation (ISP Violation) was included because it constitutes an anti-pattern that is believed to have negative effects on maintainability [32] and therefore can be considered a code smell. Borland Together is able to detect violations of this design principle.

3) Identification of Problems and Problematic Artifacts: The aim of the study was to explore situations where maintenance problems occurred due to the interaction of several code smells. Therefore, we needed to identify the artifacts that caused problems during maintenance and to record the nature of the problems caused by these artifacts. In the context of this study, we defined a maintenance-related problem as "*any struggle, hindrance or problem developers encountered while they performed their maintenance tasks that was observed by us through daily interviews and think-aloud sessions.*" The daily interviews with each developer allowed us to record the problems encountered during maintenance while they were still fresh in the developers' minds. The following is an example comment of a developer who complained about the complexity of a piece of code: "*It took me three hours to understand this method...*" We tagged such comments as partial evidence for maintainability (understandability) problems in the artifact that included this method.

During the think-aloud sessions, the developers' screens were recorded with a ZD Soft Screen Recorder.[8] Sometimes the maintenance problems were derived from more than one data source (e.g., by a combination of direct observation,

[4] http://www.genuitec.com
[5] http://tomcat.apache.org
[6] http://trac.edgewall.org
[7] http://subversion.apache.org
[8] http://www.zdsoft.com

TABLE III
CODE SMELLS AND THEIR DESCRIPTIONS (BASED ON [1, 32])

Code Smell (ID)	Description
Data Class (DC)	Classes with fields and getters and setters that do not implement any function in particular.
Data Clump (CL)	Clumps of data items that are always found together whether within classes or between classes.
Duplicated code in conditional branches (DUP)	The same or similar code structure repeated within the branches of a conditional statement.
Feature Envy (FE)	A method that seems more interested in another class other than the one it actually is in. Fowler recommends putting a method in the class that contains most of the data required by the method.
God Class (GC)	A class that takes too many responsibilities relative to the classes with which it is coupled. The God Class centralizes the system functionality in one class, which contradicts decomposition design principles.
God Method (GM)	A class has the God Method smell if at least one of its methods is very large compared to the other methods in the same class. The God Method centralizes the class functionality in one method.
Misplaced Class (MC)	A class that needs the classes from other packages more than those from its own package.
Refused Bequest (RB)	Subclasses do not want or need everything they inherit.
Shotgun Surgery (SS)	A change in a class results in the need to make a lot of little changes in several classes.
Temporary variable used for several purposes (TMP)	Temporary variables used in different contexts, implying that they are not consistently used. These types of variables can lead to confusion and the introduction of faults.
Use interface instead of implementation (IMP)	Castings to implementation classes should be avoided and an interface should be defined and implemented instead.
Interface Segregation Principle Violation (ISPV)	The dependency of one class upon another one should depend on the smallest possible interface. Even if there are objects that require non-cohesive interfaces, clients should see abstract base classes that are cohesive. Clients should not be forced to depend on methods they do not use, since this creates coupling.

the developers' statements on a given topic/element, and the time/effort spent on an activity). When it was possible to map the identified maintenance problems to an artifact, we categorized the artifact as problematic. While the assessment of problematic artifacts can be subjective to some extent, we perceived the connections between problems and code in this study to be quite direct, i.e., asserting a connection did not require much judgment that could introduce a bias.

The researcher that was on site during the study kept a logbook during the interviews and think-aloud sessions where the maintenance problems were documented in detail. For each identified maintenance problem, the following information was recorded:

- the developer and the system
- the statements provided by the developers related to the maintenance problem
- the source of the problem (e.g., infrastructure, developer, source code, external web services)
- artifacts such as classes or interfaces, and possibly methods, related to the problem

In short, the categorization of the problematic artifacts was based on either the direct observation of the developers' behavior during the think-aloud sessions, or on the comments made by the developers during the daily interviews. A detailed account on the identification of problems and problematic artifacts can be found in [35, 36].

4) Analysis Technique: To observe patterns of co-located code smells, a principal component analysis (PCA) using orthogonal rotation (varimax) was conducted on the set of artifacts that was modified/inspected by at least one of the developers during maintenance. Subsequently, a follow-up qualitative analysis based on the data from the interviews and the think-aloud sessions was performed. This analysis is based on the *explanation building technique* [37] and aims to determine the extent to which the presence of a *single code smell* or *several code smells* contributed to the problems experienced by the developers during maintenance. An essential input to the qualitative analysis was the analysis of Java files that were modified or inspected during the maintenance work. These files were identified using the logs generated by Mimec [31]. This plug-in recorded not only the type of action performed by the developer in the IDE but also the Java element (if any) that was the subject of the interaction, such as the name of the file selected or the name of the class/method being edited. For more details on the analysis of the Mimec logs we refer to [36].

IV. RESULTS

A. Exploration of the Maintenance Problems

Most of the maintenance problems that we identified were related to one of the following: (1) introduction of defects as result of changes (25%), (2) time-consuming changes (39%), and (3) troublesome program comprehension and information searching (27%). Table IV provides a description of each type of problem. The remaining problems (9%) manifested at much lower scale, and include: cumbersome configuration (6%) and debugging (3%). In total, 137 different maintenance problems were identified. Out the total number of maintenance problems, 64 (47%) related to Java source code. The remaining 73 (53%) constituted problems that were not directly related

TABLE IV
DESCRIPTION OF THE THREE MAIN TYPES OF PROBLEM IDENTIFIED

No.	Type of Problem	Description
1.	Introduction of defects	Undesired behavior or unavailability of functionality in the system (i.e., defects) that manifested after modifying different components of the system. This problem introduced delays in the work and forced the developers perform lengthy debugging or to roll back initial strategies for solving the tasks.
2.	Time-consuming or costly changes	Time consuming or costly changes were associated with situations with one of the following: i) a high number of components in the system required changes in order to accomplish a task, and, ii) cognitively demanding tasks due to the nature of the problem to be solved or due to the system's intricate design, visualization, or information distribution.
3.	Troublesome program comprehension and information searching	This problem type comprised three situations: i) the developers struggled to get an overview of the system or to obtain high-level understanding of the system's behavior, ii) the developers became confused during low-level understanding of the code because they found inconsistent or contradictory evidence in different components of the system, and iii) the developers faced time-consuming or troublesome information or "task context" searches (e.g., finding the appropriate place to make the changes, finding the data needed to perform a task).

to code (e.g., a lack of adequate technical infrastructure, developer coding habits, external services, run-time environment, and defects initially present in the system). The high percentage of non-source code-related problems suggests that problems identifiable via current definitions of code smells may only cover a smaller part (in this case, 47%) of the total problems identified during maintenance. In total, 301 artifacts (i.e., Java files) across all four systems were modified or inspected by at least one developer during maintenance, from a total of 377 artifacts. This indicates that nearly 80% of the artifacts across all four systems were modified/inspected. Out of those artifacts that were modified/inspected, 61 (20%) of them were reported as problematic during maintenance by at least one developer. Table V presents the numbers and proportions of problematic and non-problematic artifacts across the four systems, and the percentage of the total number of files that were modified/inspected in each system.

B. The Factor Analysis

A principal component analysis (PCA) was conducted on the 301 data points using orthogonal rotation (varimax). The *Kaiser-Meyer-Olkin* (KMO) measure was used to verify the sampling adequacy for the analysis, as recommended by Field [38]. With KMO = 0.604 and all KMO values for individual items being > 0.5, the sample meets the minimum acceptance criterion defined by Kaiser [39]. In addition, Bartlett's test of Sphericity, with $\chi^2(66) = 561.252$ and $p < .001$, indicates that the correlations between the items are sufficiently large for a satisfactory application of PCA. An initial analysis was run to obtain eigenvalues for each component in the data. Five components had eigenvalues over Kaiser's criterion of 1 and in combination explained 63.5% of the variance. Table VI shows the factor loadings after rotation.

TABLE V
DISTRIBUTION AND PERCENTAGE OF PROBLEMATIC VS. NON-PROBLEMATIC ARTIFACTS

System	Problematic		Non-Problematic		Total	
A	11	20%	45	81%	56	88%
B	37	30%	88	70%	125	75%
C	3	12%	22	88%	25	86%
D	10	11%	85	89%	95	80%
Total	61	20%	240	80%	301	80%

C. Relations Between Factors and Problems

In total, five factors were identified, as shown in Table VI. Below we discuss for each factor what code smells contributed to them, the code smells' role in them, and the observed effects of those smells on maintenance problems.

Factor 1: In Table VI, the code smells God Method and God Class are the closest in this factor, followed by the code smells Temporal variable used for several purposes, Duplicated code in conditional branches, and Feature Envy. Given that the detection strategies of the first two code smells are based on size measures (see Appendix in [34]), it is natural that they would appear together. In addition, large classes often use many different variables, which increase the chances of the same *Temporary variable being used for several different purposes*. Code smells in Factor 1 may, consequently, be considered to relate to the size of the code.

The great majority of these classes contained many methods that accessed data/methods from different areas of the system (i.e., methods displaying Feature Envy). This characteristic forced the developers to examine all the artifacts called by these methods in order to first understand the behavior of the class and then to identify the task context. Faults occurred in those artifacts because developers missed areas of the code that needed to be consistently changed after changes were done on the methods displaying Feature Envy. The qualitative data pointed out that artifacts with Feature Envy and God Method were associated with time-consuming changes because they involved highly complex changes in terms of the number of changes required to complete the task and also regarding the number of elements to be considered simultaneously to complete the task. (This last case comprised a cognitively demanding task.)

One or two classes in each system "hoarded" the business logic/functionality of the system (i.e., StudyDatabase, DB, and StudyDAO, as described in [34]). They were extremely large in comparison to the rest of the artifacts of the systems, and developers frequently will commit "slips" or mistakes due to the size of these classes (mainly because it is hard to navigate across the class and keep track of the changes within the class). Some of these "hoarders" also contained Temporal variables used in different contexts. These inconsistencies in the use of different variables made the developers unsure about the purpose of the variables and the behavior of the method/class containing them. This uncertainty triggered mistakes that lead to several faults, particularly in System C. Developers would change a variable, expecting that it would be used in the same way across the class. Instead, unexpected behavior would manifest and demand debugging and refactoring.

Factor 2: ISP Violation and Shotgun Surgery belong together in a separate factor (Factor 2). This distinction indicates that they may represent (to some extent) the same construct (e.g., related to wide-spread, afferent coupling). Also, they did not seem to relate much to the size of the code (Factor 1). One critical example of program comprehension in artifacts containing ISP Violation relates to the presence of *inconsistent design* (manifested in the class StudySortBean, in System A, see [34]). A major reason why the developers found System A difficult to understand was due to inconsistent and incoherent data and functionality allocation, which was considered "not logical" by the developers. Two developers actually stated that the design "did not make sense." The class StudySortBean was initially employed as a Bean[9] to sort a given list of empirical studies and present them in a report to the user. Sometime during the initial development phase (i.e., not the maintenance phase), the StudySortBean class started to acquire more responsibilities that did not correspond to its class, and it turned into an Action file. This instance is a good example of what Martin calls the "wider spectrum of dissimilar clients" [32]. As a result, an artifact initially containing a Data Class acquired an ISP Violation. Both the data and the functionality were called from many different classes, many of them unrelated. Since the allocation of the data and functionality seemed rather arbitrary to the developers, they became confused about the rationale of such a design. This case is very interesting because the ISP Violation was not the actual cause of the problem; the real problem was the inadequate allocation of data and functionality. Nonetheless, the definition of an ISP Violation and its detection strategy could identify this particular situation. We also observed that when the developers introduced faults into artifacts displaying an ISP Violation, the consequences of these faults manifested themselves across different components that depended on them. This situation caused much of the systems' functionality to stop working after changes, and in some cases, lead to unmanageable error propagation. Additionally, when changes were introduced into the aforementioned artifacts, we observed that adaptations or amendments were needed in other artifacts depending on the ISP Violators. This requirement resulted in time-consuming change propagation and also caused the introduction of defects, as developers sometimes would miss parts of the code that needed amendments, resulting in a time-consuming and an error-prone process.

Factor 3: Data Class and Data Clump are together in one factor (Factor 3). Although most artifacts displaying Data Class also displayed Data Clump, very few of the artifacts displaying both smells were deemed to be problematic during maintenance. The difficult artifacts with Data Class in turn seemed to display more affinity with ISP Violation, which was described in the previous section. Most of the problems in the Data Class were related to Task 1, where the types for the identifiers needed to be changed from Integer to String. These errors were caused when developers would forget to update some of the Data Classes, introducing defects into the systems. One observation from artifacts constituting Data Classes is that the great majority of them displayed incoming dependencies from Feature Envy methods. Data were located on the artifacts displaying Data Class, and those were accessed by methods in the artifacts that contained most of the functionality in the systems (i.e., Feature Envious methods). This code smell relationship was mentioned earlier by Pietrzak et. al. [3] and by Lanza et. al. [4, p. 78].

Factor 4: Implementation Instead of Interface represented Factor 4. This code smell appeared very seldom in our data-set and did not relate to any of the other code smells. Out of all the problematic artifacts, only two contained this smell, and one of them was problematic only because it displayed the combination of smells described in Factor 1. Consequently, in this study, this factor could not be associated with any serious maintainability problems.

TABLE VI
FACTOR LOADINGS AFTER ROTATION

	Component				
	1	2	3	4	5
GM	.751				
GC	.730				
TMP	.687				
DUP	.595				
FE	.537				
SS		.896			
ISPV		.823			
DC			.751		
CL			.721		
IMP				.823	
RB					.822
MC					-.548
Eigenvalues	2.442	1.768	1.305	1.073	1.033
% of variance	20.350	14.731	10.875	8.942	8.607

[9] In J2EE environments, it is common to use Bean files as data transfer objects. Their counterparts, the Action files (which in turn contain the business logic) access the Bean files.

TABLE VII
TENDENCIES AND CHARACTERISTICS OF INTER-SMELL RELATIONS IDENTIFIED IN THE STUDY

Relation Nickname	Factor	Major Characteristics	Characteristic Code Smells
Hoarders	1	Indicators of high internal complexity and large size where the functionality for the element has grown out of proportion. These elements are difficult to understand and change and are prone to defects due to "slips."	God Method, God Class, and Feature Envy
Confounders	1	These smells indicate that there are two or more ambiguous contexts which can easily mislead developers in what is actually happening in a particular piece of code. As a result, they are mostly associated with defects and program comprehension issues and are often located in the same classes where the Hoarders are.	Temporal variable used for several purposes and Duplicated code in conditional branches
Wide interfaces	2	Indicators of afferent coupling dispersion. If the maintenance task requires any modification in classes displaying these smells, these changes will entail unexpected side effects due to the functional coupling dispersion. Sometimes they are found in the same class as Data Containers or Hoarders.	Shotgun Surgery and ISP Violation
Data containers	3	Indicators of elements only containing data. Feature Envy methods often have dependencies on artifacts displaying these classes. Sometimes they are found in the same class as "Wide interfaces" when the allocation of data-functionality is not optimal.	Data Class and Data Clump
Unknowns	4 and 5	Not enough data from the study, very few instances were found, and most of them were not associated with maintenance problems.	Implementation instead of interface, Misplaced Class, and Refused Bequest

Factor 5: Refused Bequest and Misplaced Class constituted the last factor where Misplaced Class displayed a negative loading in the PCA. This distinction indicated that Misplaced Class tended to be negatively associated with this factor. Positive and negative loadings can be related with the same factor. For example, in surveys, negative loadings are caused by questions that are negatively oriented to a factor. A combination of positive and negative questions is normally used to minimize an automatic response bias by the respondents [40]. After analyzing the types of problems in each of the artifacts, we concluded that none of the maintenance problems observed in our study could be associated with the *presence* of Refused Bequest or the *absence* of Misplaced Class.

D. Identified Trends and Inter-Smell Relations

The results from our PCA, analysis of the problematic artifacts, and the nature of the maintenance problems caused by them, all support our position that inter-smell relations should be analyzed alongside the individual effects of code smells. To enable a more systematic treatment, we propose a set of four inter-smell relations based on our observations and analysis. Table VII gives an overview of these relations, their associated Factors from our PCA analysis, and a description of their major characteristics. The fact that the two inter-smell relations *hoarders* and *confounders* were grouped in the same factor is not surprising if we consider the fact that artifacts which exhibit *hoarders* often also exhibit the *confounders*, so the PCA would naturally put them together.

As mentioned before, some aspects of the inter-smell relations presented in Table VII were already conjectured in earlier work by Pietrzak and Walter [3] and by Lanza and Marinescu [4]. In addition to the summary in Table VII, we present a graphical overview of the relationships in Figure 1 which complements these earlier results based on concrete empirical findings from our study.

In addition to inter-smell relations, we also observed interactions between the ISP Violation code smell and other (non-smell) types of code characteristics. Moreover, we identified a critical case where interactions were observed between code smells that were distributed across coupled artifacts. We refer to these as *"coupled smells"* to distinguish them from the *co-located smells* discussed above. These cases are discussed in more detail in the next two subsections.

E. Interaction Between Smells and Other Characteristics

The first case was observed in System B, and it was related to time-consuming changes and the introduction of defects after a developer's initial changes. All developers who worked on System B reported the exact same problem: In order to complete Task 1, which consisted of modifying the functionality that accesses external data, the developers wanted to replace two interfaces (*Persistable* and *PersistentObject*[10]) with one new interface to support a String ID type. The external data employs String type identifiers as opposed to the Integer types used in the system. Replacing the interfaces was not possible

[10] These artifacts constituted implementations of the Persistence Framework. Persistence Framework is used as a part of Java technology for managing relational data (more specifically, data entities).

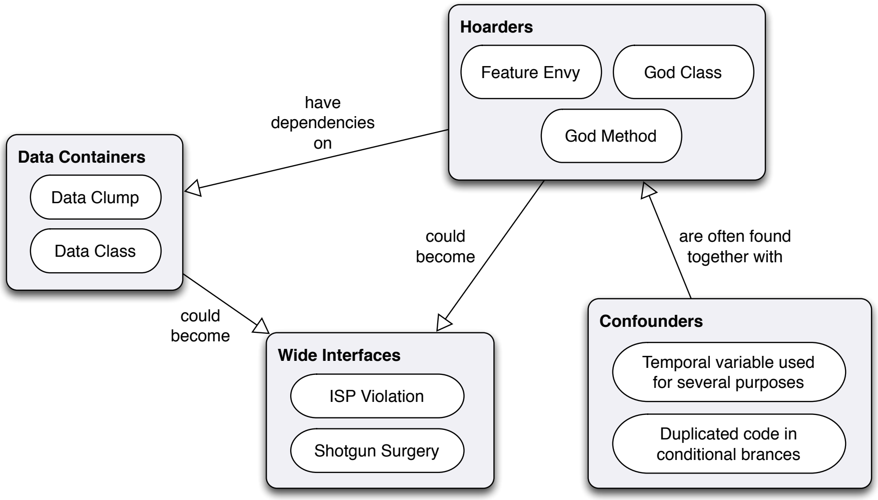

Fig. 1. Diagram displaying code smell relationships based on the observations in the study

since the entire logic flow was based on primitive types instead of domain entities. Both interfaces were restrictive and were made under the assumption that the identifiers for objects would always be Integers and thus, accessor methods getId() and setId() with a parameter of Integer type were defined. Notice that these interfaces did not display any code smells. The maintenance problems occurred because several critical classes in the system implemented these two interfaces. Many of the classes that used these interfaces exhibited ISP Violation smells, which resulted in extensive ripple effects when modifying the interfaces. After the developers modified the interfaces, an extremely high number of compilation errors were found. These errors induced the developers to rollback the initial changes in those artifacts (i.e., keep the interfaces untouched) and instead use explicit (forced) type-casting wherever a String type identifier was required. Most developers used a considerable amount of time trying to replace the interface, and they were required to rollback and perform the explicit casting. This case is an example of how the presence of a code smell may intensify or spread the effects of bad/limited design choices throughout the system. Since classes with wide afferent coupling (and therefore exhibiting the ISP Violation smell) depended on these suboptimal interfaces, any change to the interfaces resulted in a considerable ripple effect, effectively negating the abstraction benefits of using an interface.

F. Smell Interaction Across Coupled Artifacts

The second case relates to the observation that all systems except for System B contained one single class that hoarded most of the logic and functionality in those systems. These classes were very large in comparison to other classes in the system, displayed a wide spread of both afferent and efferent coupling, and demanded high amounts of changes.

All three "hoarders" exhibited ISP Violation because they displayed many incoming dependencies from different segments of the system. Because of their high level of efferent coupling, they also contained Feature Envy. These "hoarders" also exhibited the God Method smell, which is commonly present in big, complex classes. Changes in these "hoarders" were essential given the maintenance tasks, and they were time-consuming since the developers first had to understand the logic they contained. The developers reported that they found it difficult to foresee the consequences of changes given the combination of their internal complexity and the high number of dependent classes. Moreover, after the changes were made, errors would manifest in different areas of the system, causing further delays to the project. An interesting observation was that in System B, the combination of code smells present in the "hoarders" of Systems A, C, and D was not located in one artifact but instead they were distributed across several problematic artifacts. The artifacts StudySearch.java and MemoryCachingSimula.java were internally complex, and ObjectStatementImpl.java and Simula.java displayed the highest incoming dependencies. Both pairs of artifacts were strongly coupled (i.e., StudySearch.java had dependencies on ObjectStatementImpl.java, while MemoryCachingSimula.java was dependent on Simula.java). We found that the interactions between such *coupled smells* had similar effects as when code smells were *co-located* in the same artifact. Table VIII gives an overview of these coupled smells (the smell abbreviations used are explained in Table III).

G. Implications for Research and Practice

In this study, we have discovered how code smells that appear together in the same artifact can interact with each other, causing various types of maintenance problems. Practitioners could use the descriptions of the Factors identified in our study to identify critical artifacts that may need to be prioritized for refactoring. From a research perspective, we know only of one empirical study (by Abbes et al., [26]) that reports on the interaction effects between two concrete code smells (i.e., between God Class and God Method), and we believe that

TABLE VIII
OVERVIEW OF THE HOARDERS IN SYSTEM B AND THE DISTRIBUTION OF THE INDIVIDUAL SMELLS ACROSS TWO COUPLED ARTIFACTS

Filename	Individual Code Smells	Coupled Smells
StudySearch.java	GC, GM, FE	FE, GM, ISPV, GC, SS
ObjectStatementImpl.java	ISPV, SS	
MemoryCachingSimula.java	GC, TMP	ISPV, GC, SS, TMP
Simula.java	ISPV, SS	

the results from our study both extends and aligns with their findings. In addition, our findings provide concrete empirical evidence for some of the smell relations conjectured earlier by Pietrzak and Walter [3] and by Lanza and Marinescu [4].

Based on our findings, we argue that studies into the effects of inter-smell relations are a topic that deserves more attention. This position is further supported by the observation that in some large classes, the maintenance problems were not so much caused by the complexity that followed from the actual size of the class but rather were a result of interaction effects between different code smells that appeared together in that class. This distinction implies that the currently common approach for code smell detection and analysis, which is based on analyzing individual smells and not the effects of smell combinations, severely limits the capability to explain or predict maintenance problems.

Another limitation of the current approaches for code smell analysis is that the coupling between code elements that contain code smells is not considered in the analysis. However, the results from our study indicate that, from a practical perspective, interaction effects between code smells that are distributed across coupled artifacts have the same consequences as interaction effects between code smells that are co-located in the same artifact. This finding has considerable implications for further studies on code smells, since it means that, to get a more complete understanding of the role of code smells in software maintenance, dependency analysis should be included in the code smell analysis process.

H. Threats to Validity

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

1) Construct Validity: The definition of a "maintenance problem" may have been interpreted differently amongst different developers and the researcher who conducted the data collection (the first author of the paper). The code smells were automatically identified via tools to avoid subjective bias. Nevertheless, the implicit choice for the detection strategies implemented in these tools could be a potential threat to validity. Other code smell detection tools could employ different detection strategies than the ones used in this study, which in turn might lead to variation in the smells that are detected in the given subject systems.

2) Internal Validity: It is possible that some developers were more open about the problems they faced than others, and that some developers did not disclose all problems they experienced. In addition, developers may have worked more mindfully because of the interaction with researchers (Hawthorne effect). These are common threats whenever qualitative data is used in empirical studies. We have addressed these threats by using three independent collection methods (interviews, direct observation, and think-aloud sessions) and triangulating the data.[11] The detection of smells (on pre-maintenance versions of the code) was delayed until after maintenance to avoid influence and bias by the researcher collecting the data. The design of the study allowed for observation of several cases where different developers worked on the same system. This enabled an iterative explanation building process, where we could revise (confirm or reject) observations based on several cases involving the same systems. Together with a clear chain of evidence and a well-defined protocol [36], this reduced the threats typically associated with using explanation building, such as losing of focus from the original goals [37, p. 122].

3) External Validity: Our results are contingent on the contextual properties of the study and are mainly valid for maintenance projects in contexts similar to ours. The maintenance work involved medium-sized, Java-based, web-applications, and the programmers completed the tasks individually, i.e., not in teams or using pair programming. This last characteristic can affect the applicability of the results in highly collaborative environments. Given the size of the tasks and the 4-week maintenance period covered in our study, we cannot claim that our results fully represent long-term maintenance projects with large tasks. However, the tasks involved do resemble, for example, backlog items in a single sprint or iteration in the context of agile development. In this study, we closely observed the entire maintenance process for a period of four full-working weeks. We are unaware of other work that reports on experimental studies of code smells in *in-vivo* maintenance tasks that lasted longer than 240 minutes.

V. CONCLUSIONS AND FUTURE WORK

This paper reports on an empirical study that investigated inter-smell relations and their effects on the incidence of maintenance problems. By analyzing how professional developers conducted tasks on four different systems, we found empirical evidence that certain inter-smell relations were associated with problems during maintenance and also that some inter-smell

[11] The term triangulation (or "cross examination") means that multiple sources of evidence were used to validate the consistency and reliability of the results [37, pp. 97–99].

relations manifested across coupled artifacts. Our study constitutes a realistic maintenance project, and we believe that the maintenance problems that were observed are representative of those experienced in an industrial setting. Therefore, our results provide empirical evidence to guide the focus on design aspects that can be used for detecting and avoiding maintenance problems. Further studies on the basis of our findings and experiences should include the following: (i) an analysis of inter-smell relations in larger systems involving different maintenance scenarios, (ii) an analysis of the interaction effects between smells and other design properties, and (iii) an inter-smell analysis incorporating a dependency analysis to consider "coupled smells."

Acknowledgments

The authors thank Gunnar Bergersen for his support in selecting the developers and Hans Christian Benestad for providing technical support in the planning stages. They also thank Bente Anda and Dag Sjøberg for finding the resources to conduct this study and for insightful discussions. Finally, they thank the anonymous reviewers for valuable comments that helped to further improve this paper. This work was partly funded by Simula Research Laboratory and the Research Council of Norway through the projects AGILE (#179851/I40), TeamIT (#193236/I40), and InspectIT (#191171/V30).

References

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

[2] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *IEEE Int'l Conf. Softw. Maintenance (ICSM)*. IEEE, 2012, pp. 306–315.

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

[4] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer, 2005.

[5] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Working Conf. Reverse Eng. (WCRE)*, 2001, pp. 97–106.

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

[7] W. C. Wake, *Refactoring Workbook*. Addison-Wesley, 2003.

[8] R. Marinescu and D. Ratiu, "Quantifying the quality of object-oriented design: the factor-strategy model," in *Working Conf. Reverse Eng. (WCRE)*. IEEE, 2004, pp. 192–201.

[9] R. Marinescu, "Measurement and quality in object-oriented design," in *IEEE Int'l Conf. Softw. Maintenance (ICSM)*, 2005, pp. 701–704.

[10] N. Moha, Y.-G. Guéhéneuc, and P. Leduc, "Automatic generation of detection algorithms for design defects," in *IEEE/ACM Int'l Conf. on Automated Softw. Eng.*, 2006, pp. 297–300.

[11] N. Moha, "Detection and correction of design defects in object-oriented designs," in *ACM SIGPLAN Conf. Object-oriented programming, systems, languages, and applications (OOPSLA)*, 2007, pp. 949–950.

[12] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, and L. Duchien, "A domain analysis to specify design defects and generate detection algorithms," in *Fundamental Approaches to Softw. Eng.*, 2008, pp. 276–291.

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

[14] E. H. Alikacem and H. A. Sahraoui, "A Metric Extraction Framework Based on a High-Level Description Language," in *IEEE Int'l Conf. Source Code Analysis and Manipulation (SCAM)*, 2009, pp. 159–167.

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

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

[17] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, L. Duchien, and A. Tiberghien, "From a domain analysis to the specification and detection of code and design smells," *Formal Aspects of Computing*, vol. 22, no. 3, pp. 345–361, 2010.

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

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

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

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

[22] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?" in *Working Conf. Mining Softw. Repositories (MSR)*, 2010, pp. 72–81.

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

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

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

[26] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension," in *European Conf. Softw. Maint. and Reeng.*, 2011, pp. 181–190.

[27] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," in *European Softw. Eng. Conf. and Symp. Foundations of Softw. Eng.*, 2005, pp. 187–196.

[28] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems," in *IEEE Int'l Conf. Softw. Maintenance (ICSM)*, 2010, pp. 1–10.

[29] B. C. D. Anda, D. I. K. Sjøberg, and A. Mockus, "Variability and Reproducibility in Software Engineering : A Study of Four Companies that Developed the Same System," *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 407–429, 2009.

[30] G. R. Bergersen and J.-E. Gustafsson, "Programming Skill, Knowledge, and Working Memory Among Professional Software Developers from an Investment Theory Perspective," *J. of Individual Differences*, vol. 32, no. 4, pp. 201–209, 2011.

[31] L. M. Layman, L. A. Williams, and R. St. Amant, "MimEc," in *Int'l Ws. Coperative and Human Aspects of Softw. Eng.*, 2008, pp. 73–76.

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

[33] R. Marinescu, "Measurement and Quality in Object Oriented Design," Doctoral Thesis, "Politehnica" University of Timisoara, 2002.

[34] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations in the maintainability of a system : An empirical study (Report no. 2012-14)," Simula Research Laboratory, Tech. Rep., 2012.

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

[36] A. Yamashita, "Measuring the outcomes of a maintenance project: Technical details and protocols. (Report no. 2012-11)," Simula Research Laboratory, Oslo, Tech. Rep., 2012.

[37] R. Yin, *Case Study Research : Design and Methods (Applied Social Research Methods)*. SAGE, 2002.

[38] A. Field, *Discovering Statistics Using SPSS*, 3rd ed. SAGE Publications, 2009.

[39] H. Kaiser, "An index of factorial simplicity," *Psychometrika*, vol. 39, no. 1, pp. 31–36, 1974.

[40] G. E. Dunteman, *Principal components analysis. Sage university paper series on quantitative applications in the social sciences*. Newbury Park, CA: Sage, 1989.