Exploring the impact of inter-smell relations in the maintainability of a system: An empirical study

Aiko Yamashita
Simula Research Laboratory
Department of Informatics, University of Oslo
Oslo, Norway
aiko@simula.no

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

Abstract

Code smells are indicators of deeper problems in the design that may cause difficulties in the evolution of a system. While previous studies have mainly focused on studying the effects of individual smells on maintainability, we believe that interactions tend to occur between code smells. The research in this paper investigates the potential interactions amongst twelve different code smells, and how those interactions can lead to maintenance problems. Four medium-sized systems with equivalent functionality but dissimilar design were examined for smells. The systems were the object of several change requests for a period of four weeks. During that period, we recorded on a daily basis problems faced by developers and their associated Java files. The first analysis is based on Principal Component Analysis (PCA), to identify components formed by collocated code smells (i.e., smells located in the same file). Analysis on the nature of the problems, as reported by the developers in daily interviews and think-aloud sessions, revealed how some of the collocated smells interacted with each other, causing maintenance problems. Finally, we could observe that some interactions occur across files, for which we suggest integrating dependency analysis when analyzing effects of code smells on maintainability.

1. Introduction

The presence of code smells indicate that there are issues with code quality, such as understandability and changeability, which can lead to the introduction of faults [5]. In [17], Fowler and Beck describe twenty-two code smells and associate each of them with refactoring strategies that can be applied to prevent potentially negative consequences of "smelly" code. However, code smells are only indicators of problematic code. Not all of them are equally harmful, and some may not even be harmful in some contexts. In addition, refactoring implies a certain cost and risk, e.g., any changes made in the code may induce unwanted side effects and introduce faults in the system. Consequently, we need to understand better the capability of code smells to explain maintenance problems.

When analyzing code smells, we often can observe that several code smells tend to occur together in the same file. We conjecture that interaction effects between code smells can intensify problems caused by individual code smells or lead to additional, unforeseen maintenance issues. Pietrzak introduced the notion of *inter-smell relations* in [44], where he provides examples of different inter-smell relations. For example, one type of inter-smell relation reported in [44] (p. 77) called *plain-support* is defined as: "… *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*...". Inter-smell relation has only brought to attention recently, but the concept is promising to understand the nature of code smells. Whiles Pietrzak suggested the notion of inter-smell relations to support more accurate detections of code smells [44], we suggest that the study of inter-smell relations can substantially help to the understanding of how code smells can cause problems to developers during maintenance.

This paper reports a study where we examined for the presence of twelve code smells in four medium-sized Java systems. The systems were the object of several change requests for a period

of up to four weeks. During that period, we recorded problems faced by developers and their associated Java files, on a daily basis. The nature of the problems reported by developers was also recorded in detail. The maintenance problems were identified via interviews and think-aloud sessions with the developers. Principal Component Analysis (PCA) was used to identify components representing collocated code smells (i.e., smells located in the same file). Analysis on the nature of the problems, as reported by the developers in daily interviews and think-aloud sessions, revealed how some of the collocated smells interacted with each other, causing problems to developers during maintenance activities. Moreover, we found that some interactions occur across smells located across coupled files.

The remainder of this paper is structured as follows: Section 2 presents the theoretical background and related work. Section 3 describes the study design, including a description of the systems under analysis and the maintenance tasks. Section 4 presents and discusses the results. Section 5 summarizes our findings and presents plans for future work.

2. 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 could lead to the introduction of faults [17]. Beck and Fowler [17] informally describe 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 the system [25]. Code smell analysis allows for integrating both assessment and improvement in the software evolution process.

Van Emden and Moonen [50] provided the first formalization of code smells and developed an automated code smell detection tool for Java. Mäntylä [31] and Wake [51] proposed two initial taxonomies for code smells. Mäntylä investigated how developers identify and interpret code smells, and how this compares to results from automatic detection tools. Examples of recent approaches for code smell detection can be found in [3, 21, 34, 33, 36, 37, 39, 40, 38, 47]. Automated detection has been implemented in commercial tools such as Borland Together [9] and InCode [18].

Previous studies have investigated the effects of individual code smells on different maintainability related aspects, such as *defects* [10, 19, 28, 41, 46], *effort* [2, 11, 12, 29] and *changes* [20, 22, 42]. From the empirical studies identified, only the study by Abbes et al. [2] brings up the notion of *interaction effects* across code smells. Abbes et al. [2] conducted an experiment in which 24 students and professionals were asked questions about the code in six OSSs. They concluded that classes and methods identified as God Classes and God Methods in isolation had no effect on effort or quality of responses, but when appearing together, they led to a statistically significant increase in response effort and a statistically significant decrease in the percentage of correct answers, i.e., significantly higher problems with the understanding of the code. We believe that the explanatory and predictive power of code smells can be improved by considering and investigating inter-related smells, rather than only focusing on the study of individual code smells. This study attempts first to identify tendencies with respect to collocated code smells via PCA, and subsequently explore the consequences of potential interaction effects on the incidence of problems during maintenance.

3. The Empirical Study

3.1 The Systems Maintained

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 in 2003. Based on the bids, four Norwegian consultancy companies were hired to independently develop a version of the system, all using the same requirements specification. More details on the original development projects can be found in [4]. The four development projects led to four systems with the same functionality. We will refer to the four systems as System A, System B, System C and System D in this paper. The systems were primarily developed in Java and they all have similar three-layered architectures. Although the systems comprised nearly identical functionality, there were substantial differences in how the systems were designed and coded, as it is indicated by the difference in lines of code (LOC) per system in Table 1.

Table 1: Size of the systems analyzed

System	A	B	C	D
LOC	7937	14549	7208	8293

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, in order to access data related to researchers at Simula Research Laboratory as well as information on the publications.

3.2 The Maintenance Tasks and the Developers

In 2008, Simula Research Laboratory introduced a new CMS called *Plone* [45], and consequently it was no longer possible for the systems to remain operational. This situation required the systems to be adapted to the new environment. The adaptive task, together with an additional functionality required for the systems constitute the goals for the maintenance project reported in this paper. Two Eastern Europe software companies, at a total cost of approx. 50.000 Euros, conducted the maintenance tasks between September and December 2008. The maintenance tasks are briefly described in Table 2 and were completed by six different developers. All developers completed all three maintenance tasks individually. The developers were recruited from a pool of 65 participants of a previously completed study on programming skill [8]. More about the skill scores used for this purpose can be found in [7]. All developers were evaluated to have sufficient English skills for the purpose of our study.

Table 2: Maintenance tasks

Task	Description
1. Adapting the system to the new Simula CMS	The systems in the past 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). Now Simula uses a CMS based on the Plone platform, which uses an OO database called ZODB [54]. In addition, the Simula CMS database previously had unique identifiers based on Integer type, for employees and publications. Now a Char type is used as 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 in order to access all the information associated with employees and publications.
2. Authentication through web services	Under the previous CMS, authentication was done through a connection to a remote database and used authentication mechanisms available on that time for Simula Web site. Task 2 consisted of replacing the existing authentication by calling a web service provided for this purpose.
3. Add new reporting functionality	This functionality implemented in Task 3 provided a set of options for configuring personalized reports, where 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 had been 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 by only the owner of the report configuration.

3.3 Study Design

a) 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 given the specification of the maintenance task. When needed, they would discuss the maintenance tasks with the researcher who was present at the site during the entire project duration. We had daily meetings with the developers where we tracked the progress and the problems encountered. Think-aloud sessions were conducted every second day, in a random point of the day, lasting 30 minutes. Acceptance tests and individual open interviews, with duration of 20-30 minutes, were conducted once all 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 the development tool, together with MySQL [43] and Apache Tomcat [1]. Defects were registered in Trac [14], and Subversion or SVN [16] was used as the versioning system. A plug-in for Eclipse called Mimec [27] was installed in each developer's computer, in order to log all the user actions performed at the GUI level, with milliseconds precision. After completion of the three maintenance tasks on one system, they repeated the same maintenance tasks on a second system. The developers varied with respect to which of the four systems they received as the first and the second system. Clearly, there is a learning effect from repeating the same tasks on a second system. This learning effect does, however, also reflect a quite realistic situation where the developers have much relevant experience, i.e., have completed quite similar tasks before.

b) Code smells analyzed. Twelve code smells were extracted from the systems by using Borland Together™ [9] and InCode [18]. Table 3 presents descriptions of the code smells that were detected in the systems (taken from [17, 35]). The detection strategies used in the tools are based in [32] (see Appendix A). A design principle violation called *Interface Segregation Principle Violation* (a.k.a. ISP Violation) was included (See Martin in [35]). This was included because we thought it could be an essential indicator of maintenance problems and because Borland Together was able to detect this violation. This code smell is not part of the twenty-two smells defined by Fowler and Beck, but it can be considered as a *code smell* since it constitutes an anti-pattern believed to have negative effects on maintainability [35].

Table 3: Code smells and their descriptions from [17, 35]

Code Smell (ID)	Description
Data Class (DC)	Classes with fields and getters and setters not implementing any function in special
Data Clump (CL)	Clumps of data items that are always found together whether within classes or between classes
Duplicated code in conditional branches (DUP)[1]	Same or similar code structure repeated within a the branches of a conditional statement.
Feature Envy (FE)	A method that seems more interested in another class other than the one it is actually in. Fowler recommends putting a method in the class that contains most of the data the method needs.
God Class (GC)	A class has the God Class smell if the class 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 the decomposition design principles.
God Method (GM)	A class has the God Method bad smell if at least one of its methods is very large compared to the other methods in the same class. God Method centralizes the class functionality in one method
Misplaced Class (MC)	In "God Packages" it often happens that a class 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 is	Consists of temporary variables that are used in different contexts, implying that they are not consistently

[1] Note that this smell is not the actual Duplicated Code, but a local version of it, which are only located across conditional branches. This smell was included because Borland Together could detect it. Analysis of other types of Duplicated Code is out of the scope of this study.

used for several purposes (TMP)	used. They can lead to confusion and 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 to 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

In addition to the code smells, we observed file size, measured as the number of lines of code (LOC) including comments and blank lines, and observed the size of the task (churn size) on a file. Churn size was measured as the sum of lines of code inserted, updated or deleted on a file. These variables were measured using SVNKit [49] (a Java library for requesting information to Subversion).

c) Identification of maintenance problems and problematic files. The aim of the study is to explore situations where maintenance problems occur due to the interaction of several code smells. Thus, we need to identify the files that caused problems during maintenance and record the nature of the problems caused by them. In the context of this study we interpret a maintenance-related problem as "*any struggle, hindrance or problem developers encounter, and observed by us through daily interviews and think-aloud sessions, while they performed their maintenance tasks*".

The daily interviews with each developer enabled us to record the problems encountered during maintenance while they were still fresh in their mind. The following is an example of a comment given by one developer, who complains on the complexity of a piece of code: "*It took me 3 hours to understand this method...*" We use such comments like this as evidence that there were maintenance (understandability) problems in the file that included this method.

During the think-aloud sessions, the developers' screens were recorded with ZD Soft Screen Recorder [48]. Sometimes the maintenance problems were derived from more than one data source (e.g., by a combination of direct observation, the developers' statements on a given topic/element, and the time/effort spent in an activity). When it was possible to map the identified maintenance problems to a file, we categorized that file as problematic.

An example of our process to collect and structure data related to the variable "problematic file" is given in Table 4. In this example, the observations by the researcher and the statements from the developer lead to the conclusion that the initial strategy of replacing several interfaces in order to complete Task 1 was not feasible due to unmanageable error propagation. The developer spent up to 20 minutes trying to follow the initial strategy (i.e., replace the interfaces), but decided then to rollback and to follow an alternative strategy (i.e., forced casting in several locations) instead. As a result of this information (i.e., problems due to change propagation), the files containing these interfaces were deemed as be problematic. While the assessment of problematic files to some extent can be subjective, we perceived the connections between problems and code in this study to be quite direct and not did require much problematic judgment.

Table 4: Excerpt from a think-aloud session

Code	**Statement or Action taken by Developer**	**Observation / Interpretation**
Goal	Change entities' ID type from Integer to String	This is part of the requirements in Task 1.
Finding	"Persistence is not used consistently across the system, only few of them are actually implementing this interface so…"	*Persistence*[2] is referred to as two interfaces for defining business entities, which are associated

[2] A persistence framework is used as part of Java technology for managing relational data (more specifically data entities). For more information on Java persistence, see www.oracle.com

		to a third-party persistence library, which is not used consistently in the system.
Strategy	"I will remove this dependency, I will remove two methods from the interface (getId an setId) added for integer and string. This strategy forces me to check the type of the class but this is better than having multiple type forced castings throughout the code."	Developer decides to replace two methods of the Persistence interface (i.e., getId() an setId()) which are using Integer and will replace them with methods with String parameters.
Action	Engages in the process of changing id in interface PersonStatement.java	Developer engages in the initial strategy.
Muttering	"Uh, updates? just look at all these compilation errors…"	Developer encounters compilation errors after replacing the methods in the interfaces.
Action	Fix, refactor, correct errors.	Starts correcting the errors
Strategy	"Ok… I need to implement two types of interfaces, one for each type of ID for the domain entities. I will make PersistentObjectInt.java for entities that use Integer IDs and PersistentObjectString.java for String IDs.	Change of strategy, decides to actually replace the interface instead of replacing the methods in the interface.
Action	Fix more errors from Persistable.java	More compilation errors appear
Action	Continue changing interface of the entity classes into PersistentObjectInt and PersistentObjectString	Attempt to continue with the second strategy.
Action	(After 20 minutes) Roll back the change	Developer realizes that the amount of error propagation is ummanegeable, so rollbacks the changes.
Muttering	"Hmm… how to do this?"	Developer thinks of alternative options.
Strategy	"Ok, I will just have to do forced casting for the cases when the entity has String ID"	Developer decides to use the least desirable alternative: forced type castings whenever is required.

A logbook was kept during the interviews and think-aloud sessions where the maintenance problems were registered in detail. For each identified maintenance problem the following information was extracted:

a. The developer and the system.
b. The statements provided by the developers related to the maintenance problem.
c. The source of the problem, e.g., whether it was related to the Java files, the infrastructure, the database, external services, etc.
d. List of files/classes/methods mentioned by the developer when talking about the maintenance problem.

In short, the categorization of the problematic files was based on either the direct observation of the developers' behavior in the think-aloud sessions or on the comments made by the developers during the daily interviews.

d) Analysis technique. A principal component analysis (PCA) using orthogonal rotation (varimax) was conducted on the set of files that were modified/inspected by at least one of the developers during maintenance, in order to observe patterns of collocated code smells. 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 *explanation building technique* [53] and aimed at determining 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 by using the logs generated by Mimec [26]. 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 Mimec logs, see [52].

4. Results

4.1 Exploration of the Maintenance Problems

Most of the maintenance problems identified were related to: (1) introduction of defects as result of changes (25%), (2) time-consuming changes (39%), and (3) troublesome program comprehension and information searching (27%). Table 5 provides a description of each type of problem.

Table 5: Description of the three main types of problem identified

No.	Type of problem	Description
1	Introduction of defects	Undesired behaviour, or unavailability of functionality in the system (i.e., defects) manifested after modifying different components of the system. This introduced delays in the work, and forced the developers in several cases to rollback initial strategies for solving the tasks.
2	Time-consuming or costly changes	Time consuming or costly changes were associated with situations with: 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 intrincate design, visualization or information distribution.
3	Troublesome program comprehension and information searching	This problem type comprised three situations: i) The situation where the developers struggle to get an overview of the system or to get high-level understanding of the system's behaviour, ii) The situation with low-level understanding of code where the developers become confused because they find inconsistent or contradictory evidence in different components of the system, and iii) The situations with time consuming or troublesome information or "task context" search (e.g., finding the place to perform the changes, finding the data needed to perform a task).

In total, 137 different maintenance problems[3] were identified. From the total number of maintenance problems, 64 (47%) related to Java source code. The remaining 73 (53%) constituted problems not directly related to code (e.g., lack of adequate technical infrastructure, developer coding habits, external services, runtime 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. This indicates that there are substantial limitations in the use of source code analysis to explain maintenance problems. This may also imply that alternative evaluation methods should be used in combination with code smell analysis, in order to achieve a comprehensive maintainability evaluation.

In total, 301 Java files across all four systems were modified or inspected by at least one developer during maintenance. Out of those files, 61 (approx. 20%) of them were reported as problematic during maintenance by at least one developer. Table 6 presents the numbers and proportions of problematic files across the four systems.

Table 6: Distribution and percentage of problematic vs. non-problematic files

System	Problematic = 1		Problematic = 0		N
A	11	20%	45	81%	56
B	37	30%	88	70%	125
C	3	12%	22	88%	25
D	10	11%	85	89%	95
Total	61	20%	240	80%	301

[3] A more complete description of the nature and distribution of the different problems identified during maintenance are available by sending a request to the first author.

4.2 The Factor Analysis

A principal component analysis (PCA) was conducted on the 301 data points using orthogonal rotation (varimax). The Kaiser-Meyer-Olkin measure verified the sampling adequacy for the analysis, KMO = .604, and all KMO values for individual items were > .5, which is above the acceptable limit according to Kaiser (1974). Bartlett's test of Sphericity $\chi^2(66) = 561.252$, $p < .001$, indicated that the correlations between the items were sufficiently large for 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 (See Table 7). Table 8 shows the factor loadings after rotation.

Table 7: Total Variance Explained

Component	Initial Eigenvalues			Extraction Sums of Squared Loadings			Rotation Sums of Squared Loadings		
	Total	Var. %	Cum. %	Total	Var. %	Cum. %	Total	Var. %	Cum. %
1	2.442	20.350	20.350	2.442	20.350	20.350	2.315	19.291	19.291
2	1.768	14.731	35.081	1.768	14.731	35.081	1.693	14.108	33.399
3	1.305	10.875	45.956	1.305	10.875	45.956	1.462	12.180	45.579
4	1.073	8.942	54.898	1.073	8.942	54.898	1.098	9.147	54.725
5	1.033	8.607	63.505	1.033	8.607	63.505	1.054	8.779	63.505
6	.885	7.378	70.883						
7	.826	6.881	77.764						
8	.767	6.389	84.153						
9	.650	5.415	89.568						
10	.554	4.613	94.181						
11	.406	3.385	97.566						
12	.292	2.434	100.000						

Table 8: 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

4.3 Relations between Factors and Maintenance Problems

In total, five factors are identified through PCA, as depicted in Table 8. In this section, we will discuss for each factor, the code smells part of them, the Java files displaying the some of the combinations of the code smells, and report the observed effects of the code smells on the incidence of maintenance problems.

Factor 1. In Table 8, we see that 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 A), it is natural that they appear together. Also, large classes often use many different variables, which increase the chances of the

presence of Temporary variable used for several purposes. Feature Envy is also present when there are complex methods (i.e. God Methods) that need many parameters from other classes. Code smells in Factor 1 may, consequently, be considered to relate to the size of the code.

Table 9 displays the system, and code smells of the files deemed problematic during maintenance that contained at least one instance of the God Method smell (as this last one represents best the Factor 1). 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, or FE). This characteristic forced the developers to examine all the files called by these methods in order to first understand the behaviour of the class, and to identify the pertinent data or location where to perform the changes. Faults occurred in those files because developers missed areas on the code that needed to be consistently changed after changes were done on the classes displaying the efferent coupling. Also, files with Feature Envy and God Method were associated to time-consuming changes. This was because they demanded highly complex changes, in terms of number of changes required to complete the task, and in terms of the number of elements that a developer needs to consider simultaneously in order to complete the task (this last case comprising a cognitively demanding task).

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

Table 9: Problematic files containing at least one God Method smell

File	System	DC	CL	DUP	FE	GC	GM	ISPV	MC	RB	SS	Temp	Imp
StudyDatabase	A	0	0	0	**7**	0	**1**	**1**	0	0	**1**	**1**	**1**
PrivilegesManageAction	B	0	0	0	0	0	**1**	0	0	0	0	**1**	0
StudiesEditAction	B	0	0	0	0	**1**	**2**	0	0	0	0	0	0
StudiesSearchAction	B	0	0	0	**1**	0	**1**	0	0	0	0	**1**	0
StudySearch	B	0	0	0	**2**	**1**	**2**	0	0	0	0	0	0
DB	C	0	0	**2**	**16**	**1**	**2**	**1**	0	0	0	**1**	0
StudyDAO	D	0	0	**1**	**10**	**1**	**2**	**1**	0	0	**1**	**1**	0

Factor 2. ISP Violation and Shotgun Surgery belong together in a separate factor (Factor 2). This indicates that they may represent, to some extent, the same construct (e.g., related to wide-spread, afferent coupling). Also, they do not seem to relate much to the size of the code (Factor 1). Table 10 displays the code smells for the files that displayed the ISP Violation. As can be seen, a high proportion of problematic files with solely the ISP Violation and the Shotgun Surgery (7 out of 12). One critical example of program comprehension in files containing ISP Violation relates to the presence of *inconsistent design* (manifested in the class StudySortBean, in System A). A major reason why the developers found System A difficult to understand seems to be due to inconsistent and incoherent data and functionality allocation, which was considered 'not logical' by the developers (two developers literally stated that the design "did not make sense"). The class

StudySortBean was initially employed as a Bean[4] to sort a given list of empirical studies and present them in a report to the user. Probably throughout the initial development phase (i.e., not the maintenance phase), StudySortBean class started to acquire more responsibilities that did not correspond to the class, and turned into an Action file. This would be a good example of what Martin [35] calls, "wider spectrum of dissimilar clients..". This file was originally a Bean file that should only contain data, but it ended up containing functionality. As a result, this file initially containing Data Class, acquired the 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 got confused about the rationale of such design. This case is very interesting because ISP Violation is not the real cause of the problem (the real problem was the inadequate allocation of data and functionality); nonetheless, the definition of ISP Violation and subsequent detection strategy could identify this situation.

Table 10: Problematic files containing ISP Violation smell

File	System	DC	CL	DUP	FE	GC	GM	ISPV	MC	RB	SS	Temp	Imp
StudyDatabase.java	A	0	0	0	**7**	0	**1**	**1**	0	0	**1**	**1**	**1**
StudySortBean.java	A	**1**	**1**	0	0	0	0	**1**	0	0	0	0	0
ObjectStatementImpl.java	B	0	0	0	0	0	0	**1**	0	0	**1**	0	0
Person.java	B	0	0	0	0	0	0	**1**	0	0	**1**	0	0
Simula.java	B	0	0	0	0	0	0	**1**	0	0	**1**	0	0
Table.java	B	0	0	0	0	0	0	**1**	0	0	**1**	0	0
DB.java	C	0	0	**2**	**16**	**1**	**2**	**1**	0	0	0	**1**	0
Nuller.java	D	0	0	0	0	0	0	**1**	0	0	**1**	0	0
StudyDAO.java	D	0	0	**1**	**10**	**1**	**2**	**1**	0	0	**1**	**1**	0
StudySDTO.java	D	**1**	0	0	0	0	0	**1**	0	0	0	0	0
WebConstants.java	D	0	0	0	0	0	0	**1**	0	0	**1**	0	0
WebKeys.java	D	0	0	0	0	0	0	**1**	0	0	**1**	0	0

We also observed that when the developers introduced faults in files displaying 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. Also, when changes were introduced to the abovementioned classes, we observed that adaptations or amendments were needed in other classes depending on the ISP Violators. This resulted in time-consuming change propagation. This situation 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 files displaying Data Class also displayed Data Clump, very few of the files displaying both smells were deemed as problematic during maintenance. The problematic files with Data Class in turn seemed to display more affinity with ISP Violation, as described in the previous section. Most of the problems in Data Class related to Task 1, where the types for the identifiers needed to be changed from Integer to String, and developers would forget to update some of them, introducing defects to the systems. One observation from these files is that the great majority of them displayed incoming dependencies from Feature Envy methods. Data was located on files displaying Data Class, and those were accessed by methods in the classes that contained most functionality in the systems. This code smell relation has been mentioned by Pietrzak et.al. [44] and Lanza et.al. (p.78)[24].

[4] 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 11: Problematic files containing Data Class smell

File	System	DC	CL	DUP	FE	GC	GM	ISPV	MC	RB	SS	Temp	Imp
StudyMaterialForm	A	**1**	0	0	0	0	0	0	0	**1**	0	0	0
StudySortBean	A	**1**	**1**	0	0	0	0	**1**	0	0	0	0	0
ObjectJoinableStatementImpl	B	**1**	0	0	0	0	0	0	0	0	0	0	0
Privilege	B	**1**	0	0	0	0	0	0	0	0	**1**	0	0
PrivilegePersonsRelStatement	B	**1**	0	0	0	0	0	0	0	0	0	0	0
PrivilegesForm	B	**1**	0	0	0	0	0	0	0	**1**	0	0	0
Publication	B	**1**	0	0	0	0	0	0	0	0	0	0	0
StudyPersonRelStatement	B	**1**	0	0	0	0	0	0	0	0	0	0	0
StudySearchForm	B	**1**	0	0	0	0	0	0	0	**1**	0	0	0
StudyStatement	B	**1**	0	0	**1**	0	0	0	0	0	0	0	0
StudySDTO	D	**1**	0	0	0	0	0	**1**	0	0	0	0	0
StudyMaterialForm	A	**1**	0	0	0	0	0	0	0	**1**	0	0	0
StudySortBean	A	**1**	**1**	0	0	0	0	**1**	0	0	0	0	0
ObjectJoinableStatementImpl	B	**1**	0	0	0	0	0	0	0	0	0	0	0

Factor 4. Implementation instead of interface represented Factor 4. This code smell appeared very seldom in our dataset and did not relate to any of the other code smells. From the problematic files, only two files contained this code smell, and one of the files displayed the combination of smells described in Factor 1. Consequently, in this study this code smell did not represent any maintenance problems nor displayed any relationship to other code smells.

Factor 5. Refused Bequest and Misplaced Class constitute the last factor, where Misplaced Class has a negative loading. This indicates that Misplaced Class tends to be negatively associated with this factor. Positive and negative loadings can be associated 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 [13]. By observing the types of problems in each of the files, none of them could be associated to the presence of Refused Bequest or the absence of Misplaced Class.

Table 12: Problematic files containing Refused Bequest smell

File	System	DC	CL	DUP	FE	GC	GM	ISPV	MC	RB	SS	Temp	Imp
DesAction	A	0	0	0	0	0	0	0	0	**1**	1	0	0
PerformSearchStudiesAction	A	0	0	0	**1**	0	0	0	0	**1**	0	0	0
StudyMaterialForm	A	**1**	0	0	0	0	0	0	0	**1**	0	0	0
PrivilegesForm	B	**1**	0	0	0	0	0	0	0	**1**	0	0	0
StudySearchForm	B	**1**	0	0	0	0	0	0	0	**1**	0	0	0

4.4 Trends Identified

The results from our PCA and the analysis of the problematic files and the nature of the maintenance problems caused by them support our stance that inter-smells relations should be analyzed alongside individual effects of code smells. Based on our observations, we propose an initial draft of relevant inter-smell relations and their characteristics. Several of them have already been suggested theoretically by Walter and Pietrzak in [44], and Lanza and Marinescu in [24]. Table 13 displays four major code characterizers, their associated Factors as described in our PCA analysis, and a description of each. Figure 1 is an extension of the relationship diagram proposed by Lanza and Marinescu in [24], based on our observations from study, which complements the summary provided in Table 13.

Table 13: Tendencies and characteristics of Factors identified in the study

Factor	Nickname	Major characteristic	Characteristic code smells
1	Hoarders	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, change and prone to defects due to "slips".	God Method, God Class and Feature Envy
1	Cofounders	These smells are associated mostly with defects and program comprehension issues, and often in same classes where the Hoarders are.	Temporal variable used for several purposes, Duplicated code in conditional branches
2	Wide interfaces	Indicators of afferent coupling dispersion. If the maintenance task requires any modification in classes displaying these smells, this will entail unexpected side effects due to the functional coupling dispersion. Sometimes they are found in same class as Data Containers or Hoarders.	Shotgun Surgery and ISP Violation
3	Data containers	Indicators of elements only containing data. Often Feature Envy methods have dependencies on files displaying these classes. Sometimes are contained in same class as "Wide interfaces", when the allocation of data-functionality is not optimal.	Data Class and Data Clump
4 and 5	Unknowns	Not enough data from the study, very few instances were found and most of them were not associated to maintenance problems.	Implementation instead of interface, Misplaced Class, and Refused Bequest

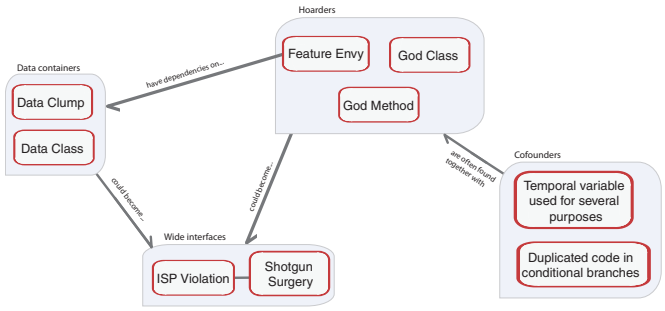

Figure 1: Diagram for displaying potential relationships from the observations in the study

During the study, we could observe the effects of interactions between code smells and also between code smells and other code characteristics. The following two observations illustrate the maintenance problems caused by interaction effects involving ISP Violation, and we believe they are representative of the types of problems identified during the study.

4.5 Interactions across code smells and other code characteristics

The first case was observed in System B, and it was related to time-consuming changes and defects after initial changes. All developers who updated System B reported this type of problem. The developers who worked on System B wanted to replace two interfaces (located in the files Persistable.java and PersistentObject.java[5]) with one new interface to support a String ID type in order to complete Task 1. Recall that Task 1 consisted of modifying functionality that accesses external data. The external data employs String type identifiers, as opposed to Integer types used in the system. Replacing the interfaces was not possible 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 defined accessor methods getId() and setId() with Integer types. Notice that these interfaces did not display any code smells. The maintenance problems seemed to occur because several critical classes in the system implemented these two interfaces. Many of the classes that implemented these interfaces

[5] These files constituted implementations of the Persistence Framework. Persistence Framework is used as part of Java technology for managing relational data (more specifically data entities). For more information on Java persistence, see www.oracle.com

(e.g., ObjectStatementImpl in Table 10) displayed ISP Violation, which resulted in extensive ripple effects when modifying the interfaces. It was observed that after the developers modified the interfaces, this led to an extremely high number of compilation errors. This induced the developers to rollback the initial changes in those files (i.e., keep the interfaces untouched) and instead perform forced casting wherever a String type identifier was required. Most developers used a considerable amount of time trying to replace the interface, and they were forced to rollback and perform the forced casting. This is an example of how the presence of a code smell may intensify or spread the effects of certain design choices throughout the system. Since classes with wide afferent coupling dispersion (and thus, containing ISP Violation) were coupled with these interfaces, any changes to the interfaces would have the same impact as if they were performed in the classes with the wide afferent coupling.

4.6 Interactions across "collocated" smells and "coupled" smells

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 (They were located in the files StudyDatabase, DB, and StudyDAO, as described in Factor 1). 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' displayed 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. They also contained God Method, which is commonly present in big, complex classes. The developers found it difficult to foresee the consequences of changes performed in the "hoarders", given the combination of their internal complexity and the high number of dependent classes. Changes in the "hoarders" were essential to the maintenance tasks, and they were time consuming since the developers first had to understand the logic they contained. Even after the changes were made, errors would manifest in different areas of the system, causing further delays to the project. One interesting observation was that in System B, the combination of code smells representing Factor 1 was not located in one file, but distributed across several problematic files. StudySearch and MemoryCachingSimula were internally complex and ObjectStatementImpl and Simula displayed the highest incoming dependencies. Both pairs of files were coupled (i.e., StudySearch had dependencies on ObjectStatementImpl while MemoryCachingSimula had dependencies on Simula). We found that the interactions between "coupled" smells had similar effects as if code smells were collocated in the same file (See Table 14).

Table 14: Hoarders in System B and how they are distributed across two coupled files

File	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	

4.7 Implications for Research and Practice

In this study, we have described how some code smells appear together in the same file and can interact with each other, causing different types of maintenance problems. Practitioners could use the descriptions of the Factors identified in this study to spot critical files that may need refactoring.

Insofar, we know only of one study (by Abbes et al., [2]) that has reported on interaction effects between code smells (i.e., between God Class and God Method), and we believe that the results from our study point into the same directions as their findings. We also provide empirical evidence of some of the inter-smell relations proposed by Pietrzak in [44], and Lanza and Marinescu in [24].

From our results, the effects of inter-smell relations seem to be a topic that deserves more attention. This stance is further supported by our observations that in some large classes, the maintenance problems may not be directly caused by the actual size of the class, but rather be a result of interaction effects across different code smells that happen to appear together in the same file. This implies that the current approach for code smell analysis (i.e., analyzing individual smells and not the effect of their combinations) limits the capability of code smells to explain much of the maintenance problems caused by design flaws.

Another limitation of the current approaches for code smell analysis is that couplings among elements containing code smells are not considered in the analyses. The findings from this study indicate that interaction effects between code smells distributed across coupled files may have the same consequences from a practical perspective as interaction effects between code smells collocated in the same file. This last finding implies a serious consideration for further studies on code smells and a need to include dependency analyses to provide a better understanding of the role of code smells in software maintenance.

4.8 Threats to validity

We consider the validity of the study presented from three perspectives:

Construct Validity. The definition of "maintenance problem" may have been interpreted differently amongst different developers and the researcher who conducted the data collection (the author of the paper). The code smells were identified via detection tools to avoid subjective bias. Nevertheless, the meaningfulness and/or common usage of the *detection strategies* used in the tools could be a potential threat. We are aware that there are other tools that can detect many of the code smells analyzed, and their detection strategies could differ to an extent from those used in this study.

Internal validity. It is possible that some developers were more open about the maintenance problems they faced than others and that some developers did not tell about all the maintenance problems they experienced. This is a common threat whenever qualitative data is used in empirical studies. Our usage of three independent collection methods, i.e., interviews, direct observation and think-aloud sessions for triangulation purposes[6] may have reduced this threat.

External validity. The results are contingent on the contextual properties of the study and the results 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 by teams or use of pair programming. This last characteristic can affect the applicability of the results in highly collaborative environments. We do not claim our

[6] In the social sciences, triangulation is often used to indicate that more than two methods are used in a study with a view to double (or triple) checking results. This is also called "cross examination".

results to fully represent long-term maintenance projects with large tasks, given the size of the tasks and the shorter maintenance period covered in our study. The tasks involved may, however, resemble backlog items in a single sprint or iteration within the context of, for example, Agile development. To the best of our knowledge we do not know of other studies reporting experimental studies on code smells on *in-vivo* maintenance tasks for more than 240 minutes, whereas in this study, we could observe closely the whole maintenance process for a period up to four full-working weeks.

5. Conclusion and Future Work

Our study constitutes a realistic maintenance project and we believe that the maintenance problems identified are representative of those experienced in several industry settings. Thus, our results may provide empirical evidence to guide the focus on design aspects that can be used for detecting and avoiding maintenance problems. The research aimed at identifying potential inter-smell relations and their effects on the incidence of maintenance problems. We found evidence that some inter-smell relations are associated with problematic files during maintenance, and that some inter-smell relations manifest across coupled files. Further studies on the basis of our findings and experiences with the reported study include code smell analyses that: i) Include larger systems and different maintenance tasks, and ii) Focus on the interaction effect between smells and other design properties, and to incorporate the analysis of coupled smells.

Appendix A

Code smells	**Detection Strategy / Metrics**	
Data class	WOC lower 33 and (NOPA higher 5 or NAM higher 5)	Weight Of Class (WOC) Number Of Public Attributes (NOPA) Number of Accessor Methods (NAM)
God method	(LOC top 20% except LOC lower 70) and (NOP higher 4 or NOLV higher 4) and MNOB higher 4	Lines Of Code (LOC) Number Of Parameters (NOP) Number Of Local Variables (NOLV) Max Number Of Branches (MNOB)
God class	AOFD top 20% and AOFD higher 4 and WMPC1 higher 20 and TCC lower 33	Access Of Foreign Data (AOFD) Weighted Methods Per Class 1 (WMPC1) Tight Class Cohesion (TCC)
Shotgun surgery	CM top 20% and CM higher 10 and ChC higher 5	Changing Methods (CM) Changing Classes (ChC)
Misplaced class	CL lower 0.33 and NOED top 25% and NOED, higher 6 and DD lower 3	Number Of External Dependencies (NOED) Class Locality (CL) Dependency Dispersion (DD)
Refused bequest	AIUR lower 1	Average Inheritance Usage Ratio (AIUR)
Feature envy	AID higher 4 and AID top 10% and ALD lower 3 and NIC lower 3	Access of Import Data (AID) Access of Local Data (ALD) Number of Import Classes (NIC)
Interface segregation principle (ISP) violation	(CIW top 20% except CIW lower 10) and AUF lower 50 and COC higher 3	Class Interface Width (CIW) Average Use of Interface (AUF) Clients Of Class (COC)
Data clump	*Abstract semantic graph* [30] can be analyzed to detect independent groups of fields and methods that appear together in multiple locations.	
Duplicated code in conditional branches	*Abstract syntax three* [15] can be analyzed to detect conditional statements, and this information can be combined with clone detection techniques (e.g., Baxter et al., [6]).	
Temporary variable is used for several purposes	Analysis of *abstract semantic graph* can be combined with semantic analysis (e.g., Landauer et al., [23]) to determine the location where temporal variables are defined and deter- mine differences in their	

	context of usage.
Use interface instead of implementation	*Abstract semantic graph* can be analyzed to detect castings to implementation classes.

Reference

1. *The Apache Tomcat 5.5 Servlet/JSP Container: What is a Realm?* 2010 20-05-2010]; Available from: http://tomcat.apache.org/tomcat-5.5-doc/realm-howto.html#What%20is%20a%20Realm?
2. Abbes, M., et al., *An Empirical Study of the Impact of Two Antipatterns Blob and Spaghetti Code on Program Comprehension*, in *Proc. European Conf. Softw. Maint. and Reengineering*. 2011. p. 181-190.
3. Alikacem, E.H. and H.A. Sahraoui. *A Metric Extraction Framework Based on a High-Level Description Language*. 2009.
4. Anda, B.C.D., 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., 2009. **35**(3): p. 407-429.
5. Arisholm, E., et al., *A web-based support environment for software engineering experiments.* Nordic J. of Computing, 2002. **9**(3): p. 231-247.
6. Baxter, I.D., et al., *Clone Detection Using Abstract Syntax Trees*, in *Proceedings of the International Conference on Software Maintenance*. 1998, IEEE Computer Society.
7. Bergersen, G. and J.-E. Gustafsson, *Programming Skill, Knowledge and Working Memory Among Professional Software Developers from an Investment Theory Perspective.* Journal of Individual Differences, 2011.
8. Bergersen, G.R. and J.-E. Gustafsson, *Programming Skill, Knowledge, and Working Memory Among Professional Software Developers from an Investment Theory Perspective.* J. of Individual Differences, 2011. **32**(4): p. 201-209.
9. Borland. *Together*. 2008 [cited 2008 September]; Available from: http://www.borland.com/us/products/together.
10. D'Ambros, M., A. Bacchelli, and M. Lanza, *On the Impact of Design Flaws on Software Defects*, in *Proc. Int'l Conf. Quality Softw.* 2010. p. 23-31.
11. Deligiannis, I., et al., *An empirical investigation of an object-oriented design heuristic for maintainability.* J. Syst. Softw., 2003. **65**(2): p. 127-139.
12. Deligiannis, I., et al., *A controlled experiment investigation of an object-oriented design heuristic for maintainability.* J. Syst. Softw., 2004. **72**(2): p. 129-143.
13. Dunteman, G.E., *Principal components analysis. Sage university paper series on quantitative applications in the social sciences*. 1989, Newbury Park, CA: Sage.
14. Edgewall-Software. *The Trac Project*. 2010 [cited 2010 June 16]; Available from: http://trac.edgewall.org/.
15. Fischer, G., J. Lusiardi, and J.W.v. Gudenberg, *Abstract Syntax Trees - and their Role in Model Driven Software Development*, in *Proceedings of the International Conference on Software Engineering Advances*. 2007, IEEE Computer Society.
16. Foundation, T.A.S., *Apache Subversion.* 2010.
17. Fowler, M., *Refactoring: Improving the Design of Existing Code*. Refactoring: Improving the Design of Existing Code. 1999: Addison-Wesley.
18. Intooitus. *InCode*. 2009 [cited 2010 July 5]; Available from: http://www.intooitus.com/inCode.html.
19. Juergens, E., et al. *Do code clones matter?* in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. 2009.
20. Khomh, F., M.D. Penta, and Y.-G. Gueheneuc, *An Exploratory Study of the Impact of Code Smells on Software Change-proneness*, in *Proceedings of the 2009 16th Working Conference on Reverse Engineering*. 2009, IEEE Computer Society.

21. Khomh, F., et al., *A Bayesian Approach for the Detection of Code and Design Smells*, in *Proceedings of the 2009 Ninth International Conference on Quality Software*. 2009, IEEE Computer Society.
22. Kim, M., et al., *An empirical study of code clone genealogies*, in *Proc. European Conf. Softw. Eng.* 2005. p. 187-196.
23. Landauer, T.K., P.W. Foltz, and D. Laham, *An introduction to latent semantic analysis.* Discourse Processes, 1998. **25**(2-3): p. 259-284.
24. Lanza, M., R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. 2005: Springer-Verlag New York, Inc.
25. Lanza, M., et al., *Object-Oriented Metrics in Practice*. 2005: Springer-Verlag New York, Inc.
26. Layman, L.M., L.A. Williams, and R.S. Amant, *MimEc: intelligent user notification of faults in the eclipse IDE*, in *Proc. Int'l Ws. Cooperative and human aspects of softw. eng.* 2008. p. 73-76.
27. Layman, L.M., L.A. Williams, and R.S. Amant, *MimEc: intelligent user notification of faults in the eclipse IDE*, in *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*. 2008, ACM: Leipzig, Germany.
28. Li, W. 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., 2007. **80**(7): p. 1120-1128.
29. Lozano, A. and M. Wermelinger, *Assessing the effect of clones of changebility*, in *Proc. Int'l Conf. Softw. Maint.* 2008. p. 227-236.
30. Mamas, E. and K. Kontogiannis, *Towards Portable Source Code Representations Using XML*, in *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*. 2000, IEEE Computer Society.
31. Mantyla, M., J. Vanhanen, and C. Lassenius, *A Taxonomy and an Initial Empirical Study of Bad Smells in Code*, in *Int'l Conf. Softw. Maint.* 2003, IEEE Computer Society.
32. Marinescu, R., *Measurement and Quality in Object Oriented Design*, in *Department of Computer Science*. 2002, "Politehnica" University of Timisoara.
33. Marinescu, R. and D. Ratiu, *Quantifying the quality of object-oriented design: the factor-strategy model*, in *Working Conf. on Reverse Eng. (WCRE)*. 2004, IEEE. p. 192-201.
34. Marinescu, R., *Measurement and quality in object-oriented design*, in *Int'l Conf. Softw. Maint. (ICSM)*. 2005, IEEE. p. 701-704.
35. Martin, R.C., *Agile Software Development, Principles, Patterns and Practice*. 2002: Prentice Hall.
36. Moha, et al., *DECOR: A Method for the Specification and Detection of Code and Design Smells.* IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 2010. **36**: p. 20-36.
37. Moha, et al., *From a domain analysis to the specification and detection of code and design smells.* FORMAL ASPECTS OF COMPUTING, 2010. **22**: p. 345-361.
38. Moha, N., Y.G. Gueheneuc, and P. Leduc, *Automatic generation of detection algorithms for design defects.* ASE 2006: 21st IEEE International Conference on Automated Software Engineering, Proceedings, 2006: p. 297-300.
39. Moha, N., *Detection and correction of design defects in object-oriented designs*, in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 2007, ACM: Montreal, Quebec, Canada. p. 949-950.
40. Moha, N., et al., *A domain analysis to specify design defects and generate detection algorithms.* Fundamental Approaches to Software Engineering, Proceedings, 2008. **4961**: p. 276-291.

41. Monden, A., et al., *Software Quality Analysis by Code Clones in Industrial Legacy Software*, in *Proc. Int'l Symp. Softw. Metrics*. 2002. p. 87-94
42. Olbrich, S., D.S. Cruzes, and D. Sjøberg, *Are all Code Smells Harmful? : A Study of God Classes and Brain Classes in the Evolution of three Open Source Systems*, in *Proc. Int'l Conf. Softw. Maint.* 2010. p. 1-10.
43. Oracle. *My Sql*. 2010 [cited 2010 September 21]; Available from: http://www.mysql.com/.
44. Pietrzak, B. and B. Walter, *Leveraging Code Smell Detection with Inter-smell relations*, in *Extreme Programming (XP)*. 2006. p. 75-84.
45. Plone-Foundation. *The Plone CMS/WCM*. 2010 [cited 2010 June 16].
46. Rahman, F., C. Bird, and P. Devanbu. *Clones: What is that smell?* in *Mining Software Repositories*. 2010.
47. Rao, A.A. and K.N. Reddy, *Detecting bad smells in object oriented design using design change propagation probability matrix.* Imecs 2008: International Multiconference of Engineers and Computer Scientists, Vols I and Ii, 2008: p. 1001-1007.
48. Soft, Z. *ZD Soft Screen Recorder*. 2012 [cited 2012 10-May-2012]; Available from: http://www.zdsoft.com/.
49. TMate-Sofware. *SVNKit - Subversioning for Java*. 2010 [cited 2010 June, 20]; Available from: http://svnkit.com/.
50. Van Emden, E. and K. Moonen, *Java quality assurance by detecting code smells*, in *Working Conf. on Reverse Engineering*. 2002. p. 97-106.
51. Wake, W.C., *Refactoring Workbook*. 2003: Addison-Wesley Longman Publishing Co., Inc. 235.
52. Yamashita, A., *Measuring the outcomes of a maintenance project: Technical details and protocols*. 2012, Simula Research Laboratory: Oslo. p. 1-19.
53. Yin, R., *Case Study Research : Design and Methods (Applied Social Research Methods)*. 2002: SAGE.
54. Zope, F. *ZODB - a native object database for Python*. 2010 [cited 2010 Sept. 15]; Available from: http://www.zodb.org/.