Empir Software Eng (2014) 19:1111–1143
DOI 10.1007/s10664-013-9250-3

Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data

Aiko Yamashita

Published online: 15 March 2013

Abstract *Code smells* are indicators of deeper design problems that may cause difficulties in the evolution of a software system. This paper investigates the capability of twelve code smells to reflect actual maintenance problems. Four medium-sized systems with equivalent functionality but dissimilar design were examined for code smells. Three change requests were implemented on the systems by six software developers, each of them working for up to four weeks. During that period, we recorded problems faced by developers and the associated Java files on a daily basis. We developed a binary logistic regression model, with "problematic file" as the dependent variable. Twelve code smells, file size, and churn constituted the independent variables. We found that violation of the *Interface Segregation Principle (a.k.a. ISP violation)* displayed the strongest connection with maintenance problems. Analysis of the nature of the problems, as reported by the developers in daily interviews and think-aloud sessions, strengthened our view about the relevance of this code smell. We observed, for example, that severe instances of problems relating to change propagation were associated with ISP violation. Based on our results, we recommend that code with ISP violation should be considered potentially problematic and be prioritized for refactoring.

Keywords Software maintenance · Code smells · Refactoring · Maintenance problems

Communicated by: Filippo Lanubile

A. Yamashita (✉)
Simula Research Laboratory, P.O. Box 134, Lysaker, Norway
e-mail: aiko@simula.no

A. Yamashita
Department of Informatics, University of Oslo, Oslo, Norway

1 Introduction

The presence of "smells" in the code may degrade quality attributes such as understandability and changeability and lead to a higher likelihood of introduction of defects. In short, code smells are symptoms of potentially problematic code from a software maintenance/evolution perspective. In Fowler (1999), Beck and Fowler 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 be harmful at all 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 defects in the system. Consequently, we need to better understand the capability of code smells to explain maintenance problems and to identify the code smells that are likely to be the best indicators of such problems. This would enable better prioritization of the most influential refactorings to improve the maintainability of a system.

Previous studies have investigated the relationship between different code smells and different maintenance outcomes (e.g., effort, change size and defects, see Section 2.2). A reasonable assumption in these studies is that more effort, larger changes, and increased defects would imply lower maintainability. In this paper, a different approach is followed: we use the presence of *maintenance problems* experienced by developers, as our measure of maintainability. A *maintenance problem* is interpreted here as "*any struggle, hindrance, or problem developers encountered while they performed their maintenance tasks, which resulted in delays or introduction of defects in the short or middle term during the project, and were possible to observe through different data collection methods*." This approach may have its limitations (e.g., maintenance problems may differ in their severity), but it may also provide several advantages compared to the previously used, outcome-based measures. In particular, we believe that observing maintenance problems is a more direct measure of the aim of code smells (i.e., to detect problematic code), than predicting effort, change size and defects. The descriptions of code smells in Fowler (1999) depict situations, where certain characteristics in the code make it difficult or *problematic* to understand, modify or test code; those problems may in turn have negative consequences in the effort, change size and defects. Consequently, there is a causal "step" in-between code smells and maintenance outcomes that needs to be investigated.

The goal of this study is to investigate the relationship between code smells and the incidence of *problems* during software maintenance, by attempting to answer the following research question: "Can code smells explain the incidence of problems during software maintenance?" To address this question, four Java systems were examined for the presence of twelve code smells. These 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 the associated Java files on a daily basis. The maintenance problems were recorded in detail from interviews and think-aloud sessions with the developers. A binary logistic regression model was developed with the variable "problematic file" as the dependent variable. The presence of twelve types of code smells together with other essential properties of the file and the task constituted the independent variables. Further, the qualitative data from interviews and think-aloud sessions were analyzed in order to support the results from the regression

model and to shed a light on *how* certain code smells can cause problems during maintenance.

The remainder of this paper is structured as follows: Section 2 discusses related work. Section 3 describes the study design, including a description of the systems under analysis and the maintenance tasks. Section 4 presents the results. Section 5 presents the discussion of the results. Section 6 summarizes our findings and presents plans for future work.

2 Related Work

2.1 Code Smells

Code smells have become an established concept in patterns or aspects of software design that may cause problems in the further development and maintenance of software systems (Fowler 1999; Lanza and Marinescu 2005; Moha et al. 2010). Code smells are closely related to Object-Oriented design principles, heuristics and design patterns, see for example principles and heuristic in Riel (1996) and Coad and Yourdon (1991), and design patterns in Gamma et al. (1994), Brown et al. (1998) and Larman (2004). Van Emden and Moonen (2001) provided, as far as we know, the first formalization of code smells and described a detection tool for Java programs. Most of the current detection approaches for code smells are automated. Examples of these approaches can be found in Marinescu and Ratiu (2004), Marinescu (2005), Moha et al. (2006, 2008, 2010) Moha (2007), Rao and Reddy (2008), Alikacem and Sahraoui (2009), Khomh et al. (2009) and Vetro et al. (2011). Work on automated detection of code smells has led to a range of detection tools such as Borland Together (Borland 2012), InCode (Intooitus 2012), JDeodorant (Fokaefs et al. 2007; Tsantalis et al. 2008) and iSPARQL (Kiefer et al. 2007). The analysis in this paper uses the tools Borland Together and InCode.

2.2 Empirical Studies on Code Smells

Zhang et al. (2011) conducted a systematic literature review to describe the state of art in code smells and refactoring, based on conferences and journal papers from 2000 to June 2009. They found that very few publications conducted empirical studies on code smells, and that focus is mostly centered on developing new tools and methods for supporting automatic detection of code smells.

Mäntylä et al. (2004) and Mäntylä and Lassenius (2006) conducted an empirical study of subjective detection of code smells. The study compared subjective detection with automated metrics-based detection and found that the results depend on developers' experience level. The authors reported that experienced developers identified more complex code smells and that increased experience with a module led to less reported code smells. Mäntylä (2005) also found a high degree of agreement between developers when reporting on the presence of simple code smells, but quite low agreement when asked to recommend refactoring decisions. Previous studies have investigated the effects of individual code smells on different maintainability related aspects, such as *defects* (Monden et al. 2002; Li and Shatnawi 2007; Juergens et al. 2009; D'Ambros et al. 2010; Rahman et al. 2010), *effort* (Deligiannis et al. 2003,

2004; Lozano and Wermelinger 2008; Abbes et al. 2011) and *changes* (Kim et al. 2005; Khomh et al. 2009; Olbrich et al. 2010).

D'Ambros et al. (2010) analyzed code from seven open source systems and found that neither Feature envy nor Shotgun surgery code smells were consistently correlated with defects across systems. Juergens et al. (2009) analyzed the proportion of inconsistently maintained Duplicated code in relation to the total set of duplicated code in C#, Java, and Cobol systems. They found (with the exception of Cobol systems) that 18 % of the inconsistently maintained duplicated code was associated with defects. Li and Shatnawi (2007) investigated the relationship between six code smells and class error probability in an industrial system and found the presence of Shotgun surgery to be connected with a statistically significant higher probability of defects. Monden et al. (2002) performed an analysis of a Cobol legacy system and concluded that the cloned modules were more reliable, but demanded more effort to maintain than non-cloned modules. Rahman et al. (2010) conducted a descriptive analysis and non-parametric hypothesis testing of source code and bug trackers in four systems. Their results suggest that clones tend to be less defect-prone than non-cloned code in general.

Abbes et al. (2011) conducted an experiment in which twenty-four students and professionals were asked questions about the code in six open source systems. They concluded that God classes and God methods in isolation had no effect on effort or quality of the responses, but when appearing together they led to a statistically significant increase in response effort and a statistically significant decrease in percentage of correct answers. Deligiannis et al. (2003) conducted an observational study where four participants evaluated two systems, one compliant and one non-compliant to the principle of avoiding God classes. Their main conclusion is that familiarity with the application domain plays an important role when judging the negative effects of a God class. They also conducted a controlled experiment (Deligiannis et al. 2004) with twenty-two students as participants. Their results suggest that a design without a God class will result in more completeness, correctness and consistency compared to designs with a God class.

Lozano and Wermelinger (2008) compared the maintenance effort of methods during periods when they did not contain a clone and when they did contain a clone. They found that there was no increase in the maintenance effort in 50 % of the cases. However, when there was an increase in effort, this increase could be substantial. They reported that the effect of clones on maintenance effort depends more on the areas of the system where the clones are located than the cloning itself. Khomh et al. (2009) analyzed the source code of Eclipse IDE, and they found that in general, Data classes were changed more often than non-Data classes. Kim et al. (2005) reported on the analysis of two medium-sized open source libraries (Carol and dnsjava). They reported that 36 % of the total amount of duplicated code in the system needed to be simultaneously updated as a consequence of the product evolution. The rest of the duplicated code evolved independently and did not require simultaneous updates. Olbrich et al. (2010) reported on an experiment involving the analysis of three open-source systems, where they observed that God and Brain classes were changed less frequently and had fewer defects than other classes when adjusting for differences in the class size.

From the identified empirical studies in code smells, it is possible to observe that not all code smells are equally harmful. Also, they are not consistently harmful across

studies, indicating that their effects are potentially contingent on contextual variables or interaction effects. For example, the study by Li and Shatnawi (2007) found that the presence of Shotgun surgery leads to defects. D'Ambros et al. (2010) on the other hand, found no such connection between Shotgun surgery and defects. Results from studies on duplicated code suggest that the effect of duplication depends of factors such as the programming language (e.g., results from Cobol system differed from the other types of systems in the study by Juergens et al. 2009). Similarly, results from studies on God class seem to give different results. Deligiannis et al. (2003) reported that God class indicated problems, while Abbes et al. (2011) concluded that a God class in isolation is not harmful.

2.3 Motivation of the Study Design

The design choice for this study is based on the assumption that observing the actual problems developers face during maintenance can provide a good picture of the role that code smells play in maintenance. Previous studies of code smells have mainly focused on duplicated code and a few other code smells. We aim to expand on the set of code smells studied, in order to better understand their respective explanatory capabilities. We believe that such insight can support refactoring prioritization endeavors. Our focus on problematic code may be closer to the original descriptions of code smells given by Fowler (1999), since code smell definitions are more closely associated with *problematic maintenance* rather than indirect measures such as change effort, change size, and defects. To the best of our knowledge, analyzing the connection between maintenance problems and a large set of code smells has never been conducted. To extend our ability to detect causal relationships in our study, we decided to support the quantitative results with qualitative observations. For these reasons, the study includes data from interviews, recordings of the developers' behavior and data from think-aloud sessions.

3 The Empirical Study

3.1 The Maintenance Project

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. Based on the bids, four Norwegian consultancy companies were hired to independently develop a version of the system, all of them using the same requirements specification. More details on the original development projects can be found in Anda et al. (2009). 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 study. The systems were primarily developed in Java and they all have similar three-layered architectures. Although the systems are comprised of nearly identical functionality, there were substantial differences in how the systems were designed and coded.

Table 1 shows the differences in lines of code (LOC) and number of Java files per system. The systems were all deployed in Simula Research Laboratory's Content Management System (CMS), which at that time was based on PHP and a relational

Table 1 Size of the systems maintained

	System A	System B	System C	System D
LOC	7,937	14,549	7,208	8,293
No. Java files	63	167	29	118

database system. The systems had to connect to the database in the CMS in order to access data related to researchers, studies, and publications, i.e., to extract the information needed for the purpose of keeping track of the empirical studies.

In 2008, Simula's Software Engineering Department outsourced a maintenance project in order to adapt these four Java applications to a new CMS named *Plone* (Plone Foundation 2012), which was introduced at Simula Research Laboratory the same year. This provided a unique opportunity to conduct an in-vivo maintenance study, and constitutes the context of the study reported in this paper. Three maintenance tasks were defined, as described in Table 2. Two tasks concerned adapting the system to the new platform and a third task concerned the addition of new functionality that users had requested.

The project was conducted between September and December of 2008, by two software companies in their respective premises in Czech Republic and Poland. The project involved six software developers (three from each company), and resulted in a total cost of 50,000 Euros.

Table 2 Maintenance tasks carried out during the study

No.	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 (information on employees at Simula and publications). Now Simula uses a CMS based on the Plone platform, which uses an OO database. In addition, the Simula CMS database previously had unique identifiers based on Integer type, for employees and publications; a String type is used now. 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 data associated with employees and publications.
2	Authentication through web services	Under the previous CMS, authentication was done through a connection to a remote database using authentication mechanisms available at that time for the 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	The devised functionality provided options for configuring personalized reports, where the user could choose the type of information related to a study to be included in the report, define inclusion criteria according to researchers who were in charge of the study, sort the resulting studies according to the date that they were finalized, and group the results according to the type of study. The configuration should be stored in the system's database and only be editable by the owner of the report configuration.

The six developers were selected from a pool of 65 participants from a previously completed study on *programming skill* (Bergersen and Gustafsson 2011), and we explicitly requested the two companies for those developers to conduct the maintenance. All the selected developers were evaluated to have good development skills (i.e., they all scored better than average skill). More about the skill scores used for this purpose can be found in Bergersen and Gustafsson (2011). In addition, all developers were deemed to have sufficient English skills for the purpose of the project.

3.2 Study Design

3.2.1 Assignment of Systems to Developers

Each of the six developers individually conducted all three tasks (ordered as in Table 2) on one system, and once they were done, they repeated the same tasks in a second system. This was done to collect more observations for different types of analysis, and gave us a total of 12 cases (six developers $\times$ two systems). Figure 1 describes the order in which the systems were assigned to each developer. The assignment of developers to systems was random, with control for equal representation, and maximizing contrast between the two cases handled by each developer (contrast was viewed in terms of system size, measured in LOC).

As it can be seen from Fig. 1, the developers varied with respect to which of the four systems we assigned as the first and the second system (designated as "first round" and "second round" systems, respectively). It can be assumed that there is a learning effect from repeating the same tasks on a second system. This learning effect does, however, also reflect quite a realistic situation where the developers have relevant experience, i.e., have completed quite similar tasks before. This means that we study the effect of code smells on maintenance problems on a mixture of tasks where the developers have and where they have not previous experience from very similar tasks.

3.2.2 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 specifications of the three maintenance tasks. When needed, they would discuss the maintenance tasks with the researcher (the author of this paper) who was present at the site during the entire project duration.

Daily, individual meetings (30 min approx.) were conducted with each developer and the researcher present at the site. These meetings were held mostly in the beginning of the day (unless some unexpected situation arose), and consisted of semi-structured interviews, where aspects such as: *time spent on the different tasks*, *level of completeness and difficulty of task*, *bugs and issues found* where monitored. For such

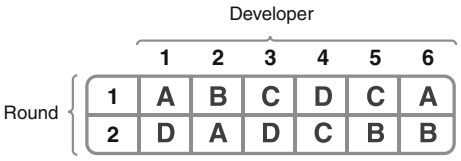

Fig. 1 Assignment of systems to developers in the case study

purposes, the researcher counted with a check-list of aspects to cover throughout the meeting, which would be asked in case the aspect was not mentioned by the developer. Thirty minute think-aloud sessions were conducted every second day, and performed at random points of the development work. Acceptance tests and individual open interviews, with a duration of 20–30 min, 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). In open-ended interviews, the researcher also counted with a check-list of aspects to be discussed, but allowing the developer to freely talk about his/her impressions. All interviews were audio-recorded, and the researcher would take notes during the meetings. For further details on the interview protocols, see Yamashita (2012b).

Eclipse was used as the development tool, together with MySQL (Oracle 2012) and Apache Tomcat (The Apache Software Foundation 2012b). Defects were registered in Trac (Edgewall-Software 2012); Subversion or SVN (The Apache Software Foundation 2012a) was used as the versioning system. A plug-in for Eclipse called Mimec (Layman et al. 2008) was installed in each developer's computer in order to log all the user actions performed at the GUI level with milliseconds precision.

A *replication package* is available in a technical report by Yamashita (2012b) where all the aspects of the maintenance project and the data collection protocol are described in detail. Also, a detailed rationale on the design of this study is available in the doctoral dissertation by Yamashita (2012a).

3.2.3 The Dependent Variable

One aim of the study was to build an explanatory model of maintenance problems. This model consisted of a dependent variable, i.e., the variable we try to explain, and independent variables, i.e., the variables used to explain the dependent variable. The *dependent variable* of our model was related to whether a Java file was perceived as problematic or not by at least one of the developers in at least one of the rounds. In the context of this study, maintenance-related problems were interpreted as "*any struggle, hindrance, or problem developers encountered while they performed their maintenance tasks, which resulted in delays or introduction of defects in the short or middle term during the project, and were possible to observe through daily interviews and think-aloud sessions.*"

The daily interviews with each developer enabled the recording of 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 complained about the complexity of a piece of code: "It took me 3 h to understand this method..." Such types of comments were used 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 a Screen Recorder program (ZD Soft 2012). Sometimes the maintenance problems were derived from more than one data source (e.g., combination of direct observation, the developers' statements on a given topic/element, and the time/effort spent on an activity). Since not all maintenance problems were associated with a piece of source code, some maintenance problems were not included in the model building. When it was possible to map the identified maintenance problems to a file, that file was categorized as problematic.

An example of the process to collect and structure data related to the variable "problematic file" is given in Table 3. 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 min trying to follow the initial strategy (i.e., replace the interfaces), but decided 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 the interfaces were deemed problematic. While the assessment of problematic files can be subjective to some extent, the connections between problems and code in this study were deemed to be quite direct. Also, during the study, a second researcher randomly selected a subset of the descriptions of the problems in the *logbook*, in order to verify if they constituted maintenance problems according to the definition agreed upon for the study.

Note that there is a many-to-many relationship between maintenance problem and files. More specifically, one maintenance problem can be associated with several or no files, and one file can be associated with different observed occurrences of maintenance problems. However, we decided to not use the *number* of associated problems of a given file to assign *weighted scores* to a file, in order to keep the analysis simple. Similarly, it could have been meaningful to classify the severity of maintenance problems. However, in many cases this would have lead to quite subjective assessments. Consequently, in order to avoid a more complex model and increased subjectivity in the interpretations, it was decided to treat "problematic file" as a binary variable.

A catalogue of maintenance problems was kept during the interviews and think-aloud sessions, in which the maintenance problems were registered in detail. For each identified maintenance problem, the following information was collected:

(a) The developer and the system.
(b) The statements given 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).
(d) List of files/classes/methods mentioned by the developer when talking about the maintenance problem.[1]

The maintenance problems were also classified according to their nature. The classification of maintenance problems was built in a bottom-up approach, inspired on *grounded theory* principles. In specific, *coding techniques* were used, following the guidelines from Edberg and Olfman (2001) and Strauss and Corbin (1998).

The identified problems were *coded* using two coding techniques, *open* and *axial coding* (Strauss and Corbin 1998). During *open coding*, the statements from the developers on each of the problems were annotated using labels (or codes) that were initially constructed from the *logbook* that the researcher kept during the project. The coding schema was revised in an iterative fashion, during the annotation process. Then, *axial coding* was used, where the annotated statements were grouped

[1]Note: one maintenance problem could be related to several problematic Java files.

Table 3 Excerpt from a think-aloud session

Code	Statement/action by developer	Observation/interpretation
Goal	Change entities' ID type from Integer to String	This is part of the requirements for Task 1.
Finding	"Persistence is not used consistently across the system, only few of them are actually implementing this interface so..."	Persistence[a] is referred to as two interfaces for defining business entities, which are associated with a third-party persistence library, and 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, 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 min) Rollback the change.	Developer realizes that the amount of error propagation is ummanegeable so rolls back 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 they are required.

[a]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

according to the most similar concepts, to derive a classification of difficulties. During this stage, we filtered out the categories that contained statements given by only one developer, and concentrated on perceptions that several developers agreed upon.

For generating the coding schema, three perspectives were used: *Activity*, *Difficulty* and *Factor*. *Activity* refers to the maintenance context in which the difficulty manifests. *Difficulty* is the manifestation of any type of hindrance to the activity being performed at the time, and *Factor* is the potential cause of the hindrance.

If we consider the following example: "...while searching of the place to make the change, I spent many hours because I had to examine many classes..." we could label the *Activity* as: "identifying the task context" (or *concept location* as defined by Rajlich and Gosavi 2004), the *Difficulty* as "time consuming or manual search", and the *Factor* as: "complexity". Examples of *Activity* labels used in the analysis are: *Change code*, *Add new code*, *Configure a library*, *Find information*, and *Debug*. Examples of some of the *Difficulty* labels are: *Error propagation*, *Confusion*, *Time consuming search*. Examples of *Factor* labels are: *DB Queries*, *Size*, *Bad naming*, *Wide-spread logic*, *Control flow*, *Duplication*, *Lack of rules*, *Data access objects*.

The coding schema was revised in an iterative fashion, during the annotation process. Then, a technique called *axial coding* was used, where the annotated statements were grouped according to the most similar concepts, using mostly the *Difficulty* perspective. In this study, the final *difficulty categories* resulting from the axial coding were: Introduction of Defects, Troublesome understanding, Troublesome learning, Troublesome Information Seeking, and Time-consuming/costly changes. The categories 2–3 were grouped into one category, as they all represent mainly program comprehension issues during maintenance.

In addition to the categorization of a file as problematic or not (and the categorization of the nature of the maintenance problem), qualitative analysis was conducted on the identified maintenance problems. This analysis was based on *explanation building technique* (Yin 2002) and aimed at determining the extent to which the maintenance problems were caused by, or only correlated with smells.

3.2.4 The Independent Variables

Twelve code smells were extracted from the systems and used as independent variables in the regression model. Table 4 presents descriptions of the code smells and their respective scale types. The choice of the twelve code smells is based on their relevance, their presence in previous code smell studies, and the availability of tools to automatically identify them. Two commercial tools: Borland Together (Borland 2012) and InCode (Intooitus 2012) were available at the time of the study, and these tools could identify these twelve smells.

The criteria for the selection of the tools was based on: (1) the fact that vendors would reveal how they detect the smells, and (2) the tools implemented the detection strategies that were comprehensively described by Marinescu (2002) and Lanza and Marinescu (2005), who proposed combinations of different metrics to detect different smells (See Appendix A for a detailed description on their detection strategies). The usage of commercial detection tools is justifiable from the perspective that is not always possible for researchers to implement their own code smell detection tools in order to conduct empirical studies.

Borland Together could also detect a design principle violation called *Interface Segregation Principle Violation* (ISP violation) (Martin 2002), and therefore this design violation was also included in the analysis. ISP Violation is not part of the twenty-two code smells defined by Beck and Fowler, but it can be considered a code smell since it constitutes an anti-pattern believed to have negative effects on maintainability (Martin 2002).

The detection tools operate at file level. According to the detection strategies implemented by these tools, there are two groups of code smells. The first group has a *binary* nature, because a *file constitutes* or *not* an instance of a code smell, such

Table 4 Code smells and their descriptions from Fowler (1999) and Martin (2002)

Code smell (ID)	Description	Variable type
Data class (DC)	Classes with fields and getters and setters not implementing any function in particular.	Binary
Data clumps (CL)	Clumps of data items that are always found together whether within classes or between classes.	Binary
Duplicated code in conditional branches (DUP)[a]	Same or similar code structure repeated within the branches of a conditional statement.	Binary
Feature envy (FE)	A method that seems more interested in others classes than the one it is actually in. Fowler recommends putting a method in the class that contains most of the data the method needs.	Continuous
God class (GC)	A class has the God Class code 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.	Binary
God method (GM)	A class has the God method code 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.	Binary
Interface segregation principle violation (ISPV)	The dependency of one class to another should consist 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.	Binary
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.	Binary
Refused bequest (RB)	Subclasses do not want or need everything they inherit.	Binary
Shotgun surgery (SS)	A change in a class results in the need to make a lot of little changes in several classes.	Binary
Temporary variable is used for several purposes (TMP)	Consists of temporary variables that are used in different contexts, implying that they are not consistently used. They can lead to confusion and introduction of defects.	Binary
Use interface instead of implementation (IMP)	Castings to implementation classes should be avoided and an interface should be defined and implemented instead	Binary

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

as the following: Data class, God class, Interface Segregation Principle violation, Misplaced class, Refused bequest, Shotgun surgery and Implementation instead of interface. The second group has a *continuous* nature, because there could be several

instances of a given smell within a file, such as: Data clump, Duplicated code in Conditional branches, Feature envy, God method and Temporary variable used for several purposes. For example, the code smells God method and Feature envy from the second group operate at method level, and since a java file may contain several methods, this implies that the tools would count the number of instances of those code smells in a given file. Table 16 in Appendix C displays the actual measurement values for each type of code smell in each of the systems.

We found that most of the smells from the second group (i.e., the smells constituting continuous variables) displayed only 1 to 2 instances per file, with the exception of Feature envy. This means that it would have not been possible to gain much in explanatory power by increasing the complexity of the model and treating these code smells as continuous variables. Consequently, all the code smells with the exception of Feature envy were treated as binary variables, i.e., code smell present = 1 and not present = 0. Natural logarithm was applied to the Feature envy variable to avoid a too strong effect from a few very high values.

In addition to the code smell variables, a variable reflecting the file size (LOC including comments and blank lines), and a variable reflecting the size of the task (churn) on a file were included. Churn was measured as the sum of lines of code inserted, updated, and deleted in a file. These variables were measured using SVNKit (TMate-Sofware 2010), a Java library for requesting information to Subversion. The variables file size and churn were included in the regression model to adjust for an increase of likelihood of a file being perceived as problematic, not because of the presence of code smells, but because of a large size or a large update of the file. Both variables were log-transformed to avoid large influences from a few very high values.

There could be differences in the effect of code smells on problematic code depending on which of the four systems were under development. Therefore, *system* was also included as a (nominal) context variable in the model. The way we defined "problematic file" (i.e., whether a Java file has been perceived as problematic or not by at least one of the developers in at least one of the rounds) implies that it is not necessary to include the variables "developer" or "round" as independent variables in the model.

3.2.5 The Analysis

An essential aspect of the study design is the notion of *task context* (i.e., files that were modified or inspected during the maintenance tasks). For the analysis, we only included the files that were modified or inspected by at least one of the developers during the maintenance project. These files were identified by analyzing the logs generated by the Mimec (Layman et al. 2008) plug-in. 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 how the Mimec logs we processed and analyzed, see Yamashita (2012b).

First, an exploration of the maintenance problems was conducted. Then, a binary logistic regression model was built using the independent variables to explain the dependent variable (i.e., likelihood of a file being problematic), including the data from all four systems. The results of the regression model were complemented by exploratory factor analysis, that is Principal Component Analysis (PCA), in order to investigate the clusters of code smells (i.e., to what degree different code smells

were correlated with others). This may be helpful to understand the nature and potential effects of clusters of code smells. Finally, a qualitative follow-up analysis was conducted based on the results from the regression analysis.

The data set used in the regression analysis was based on recording a data point each time a developer read or updated a file. The way the dependent variable was defined (i.e., that a file was categorized as problematic when at least one developer had had maintenance problems that could be related to that file), would result in a substantial amount of data having the same values for all variables except the size of the task on that file (i.e., churn) when using the original data set. To avoid this strong degree of dependency among the observations, all "duplicates" (i.e., all highly dependent observations) were removed, and the average churn of all the recordings belonging to the same file was used to represent the typical size of the update of that file. This led the data set to have a maximum of one data point per Java file, which either had the value 1 (problematic) or 0 (not problematic). Consequently, this approach increases the robustness of the model and reduces the problem of dependency between the observations.

4 Results

4.1 Exploration of the Maintenance Problems

In total, 137 different problems were identified from the different maintenance projects. From the total, 64 problems (47 %) were associated to Java source code. The remaining 73 (53 %) did not relate directly to code (e.g., lack of adequate technical infrastructure, developer's coding habits, external services, runtime environment, defects initially present in the system).

According to the categorization derived from the coding techniques, 91 % of the *total set* of identified problems related to one of three categories: (a) introduction of defects as result of changes (25 %), (b) troublesome program comprehension and information searching (27 %), and (c) time-consuming changes (39 %). The remaining 9 % were associated to cumbersome configuration (6 %), and debugging (3 %). Table 5 provides a description of the three main types of problems identified. Further details on the nature of the problems and their distribution is available in Part II, Chapter 4 of the doctoral dissertation by Yamashita (2012a).

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. The last column 'N(%)' represents the total number of files modified/inspected during maintenance and the percentage with respect to the total number of files in each system. In Table 1, we reported that System B was about twice as large as the other systems, and that the other systems had about the same number of lines of code. Table 6 shows that System B was also the system with the largest proportion of problematic files. Considering that the four systems implemented the same functionality and were subject to the same maintenance tasks, this may suggest that writing more lines of code to implement the same piece of functionality can indicate an increase in maintenance problems.

Table 5 Description of the three main types of problem identified

Type of problem	Description
Introduction of defects	Undesired behavior, or unavailability of functionality in the system (i.e., defects) manifested after modifying different components of the system. This introduced delays in the project, and forced developers to rollback initial strategies for solving the tasks on several occasions.
Troublesome program comprehension and information searching	This problem type is comprised of three situations: (1) struggle while trying to get an overview of the system or while trying to achieve high-level understanding of the system's behaviour, (2) during low-level understanding of the code, developers become confused because they find inconsistent or contradictory evidence in different components of the system, and (3) developers struggle while searching for information or the "task context" (e.g., finding the place to perform the changes and/or finding the data needed to perform a task).
Time-consuming or costly changes	Time consuming or costly changes were associated with two main situations: (1) the high number of components in the system that needed changes for accomplishing a task, made the overall task time-consuming, and (2) the presence of cognitively demanding problems to be solved, or intrincate design, visualization or distribution of information made the task difficult and, consequently, time-consuming.

Table 7 presents the percentage of files inspected or modified during maintenance that contained any of the investigated code smells. In the case of Feature Envy, since it is treated as a continuous variable, the mean and standard deviation are presented. For the actual number of code smells at system level and also within the files modified/inspected during maintenance, see Table 16 in Appendix C.

4.2 The Binary Logistic Regression Model

A binary logistic regression model was built with all the variables entered in a single step. A 50 % threshold was used for the regression model (this value is called 'Classification cutoff' in SPSS, and it was set to its default value, 0.5). The overall fit of the model is indicated by the values displayed in Table 8. The chi-square statistics of the −2 Log likelihood suggests that our model performs significantly better than the null model, i.e., a model always predicting the most common outcome. The R^2 values (Cox & Snell, and Nagelkerke) provides further support that our model has a good fit, but do also shows that, not unexpectedly, a substantial part of the variance

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

System	Problematic		Non-problematic		N (%)
A	11	20 %	45	80 %	56 (87.5 %)
B	37	30 %	88	70 %	125 (74.8 %)
C	3	12 %	22	88 %	25 (86.2 %)
D	10	11 %	85	89 %	95 (80.5 %)
Total	61	20 %	240	80 %	301 (79.8 %)

Table 7 Percentage of the files inspected or modified during maintenance that contained any of the code smells investigated

Code smells	Systems				
	A	B	C	D	All
Data class	0.18	0.2	0.32	0.22	0.21
Data clump	0.11	0.01	0.12	0.05	0.05
Duplicated code	0.02	0.02	0.04	0.02	0.02
Feature envy					
Mean	0.36	0.12	0.14	0.1	0.16
Std dev	0.501	0.37	0.578	0.36	0.423
God class	0.02	0.04	0.12	0.02	0.04
God method	0.07	0.08	0.08	0.04	0.07
ISP violation	0.11	0.06	0.04	0.12	0.09
Misplaced class	0	0.02	0	0.02	0.01
Refused bequest	0.27	0.06	0	0.01	0.08
Shotgun surgery	0.12	0.12	0	0.13	0.11
Temp. variable	0.16	0.1	0.16	0.03	0.09
Use of interface	0.05	0.02	0	0	0.02

is not explained by our model. The Hosmer and Lemeshow test gives a R^2 of 0.864, and further supports that the model provides a good fit of the data.

Table 9 shows the percentage of correctly and incorrectly classified cases from the model. The performance measures for the model are: *accuracy* = 0.847, *precision* = 0.742, and *recall* = 0.377. As can be seen, the classification performance is far from perfect, but it is much better than the random model or the null model that always classifies a file as not problematic. This provides further evidence for the meaningfulness of the proposed explanatory regression model.

Table 10 displays the independent variables of the model with their coefficients (B), standard errors (S.E.), Wald statistic (Wald), significance levels (Sig.), odds ratios (Exp(B)), and lower and upper confidence intervals for the odds ratios. All the variables for which the odds ratios are statistically significant, here defined as $p < 0.05$, are in bold and their significance levels are marked with an asterisk. The Wald statistic indicates if the coefficient is significantly different from zero. The odds ratio indicates the change in odds resulting from a unit change in the explanatory variable given that all other variables are held constant.

As can be seen in Table 10, the variable *system* is included in the model as a *context variable*. If we had used a *hierarchical logistic regression model* (Field 2009), the distinction between independent and context variable would be essential, but in our case, there is no distinction needed between context and independent, and consequently, both types of variables were included simultaneously as independent variables in the model.

Table 8 Model test

−2 Log likelihood	DF	ChiSquare	*Prob > ChiSq*
223.689	17	79.758	0.000

Cox & Snell R^2: 0.233
Nagelkerke R^2: 0.367
Hosmer & Lemeshow R^2: 0.864

Table 9 Classification performance

Observed	Classified		
	'Problematic?'		Percentage correct
	0	1	
'Problematic?'			
0	232	8	96.7
1	38	23	37.7

Table 10 indicates that a file belonging to System B, which has an odds ratio of 4.137, has a significantly higher likelihood of being connected with maintenance problems than a file belonging to the other systems. This suggests, similar to what we have observed earlier, that the size of the system might be quite important when explaining maintenance problems.

Regarding the code smells, we can see that the odds ratio for the code smell ISP violation is high (Exp(B) = 7.610, $p = 0.032$), which suggests that ISP violation is a promising code smell to explain maintenance problems. This model also find the Data clump code smell as a significant contributor to the model (Exp(B) = 0.053, $p = 0.029$), but contrary to ISP violation, this code smell indicates fewer maintenance problems.

In this study, the focus was set on identifying negative effects of code on maintenance, and not positive effects. Therefore, we did not collect much information about situations where maintenance was perceived to be easy/effortless. However, the strong, positive effect of the Data clump suggests, that this smell is a candidate for further studies, i.e., the results from this study suggests that there could be types of Data clump that increase rather than decrease the maintainability of software systems.

Table 10 Model variables

Variable	B	S.E.	Wald	Sig.	Exp(B)	95 % C.I. for Exp(B)	
						Lower	Upper
System A	0.188	0.612	0.094	0.759	1.207	0.364	4.005
System B	**1.420**	**0.481**	**8.717**	**0.003***	**4.137**	**1.612**	**10.617**
System C	−0.361	0.895	0.163	0.687	0.697	0.121	4.027
DC	0.036	0.479	0.006	0.940	1.037	0.405	2.650
CL	**−2.935**	**1.340**	**4.796**	**0.029***	**0.053**	**0.004**	**0.735**
DUP	−2.721	1.747	2.427	0.119	0.066	0.002	2.018
FE	−0.001	0.493	0.000	0.999	0.999	0.380	2.627
GC	0.605	1.180	0.263	0.608	1.831	0.181	18.494
GM	−0.810	0.916	0.782	0.377	0.445	0.074	2.679
ISPV	**2.029**	**0.948**	**4.587**	**0.032***	**7.610**	**1.188**	**48.749**
MC	1.151	1.325	0.755	0.385	3.162	0.236	42.418
RB	0.231	0.633	0.133	0.716	1.260	0.364	4.359
SS	−0.654	0.778	0.705	0.401	0.520	0.113	2.392
TMP	0.089	0.659	0.018	0.892	1.093	0.301	3.977
IMP	0.311	1.108	0.079	0.779	1.365	0.156	11.978
Churn	**0.723**	**0.152**	**22.694**	**0.000***	**2.061**	**1.531**	**2.775**
LOC	0.189	0.272	0.482	0.487	1.208	0.709	2.058
Constant	−4.237	1.162	13.305	0.000	0.014		

The coefficient of the variable churn is significantly different from zero ($p < 0.001$), and the odds ratio is higher than one. This is not surprising, given that the more work completed on a file, the more likely is it that there will be some problem connected with that file.

Multicollinearity may lower the robustness of the interpretation of the individual coefficients of our model, which does include variables likely to be correlated. However, when reducing the multicollinearity by representing each factor (see the principal component analysis in Section 4.3) by one of its code smells, the coefficients did not change very much. Also, multicollinearity diagnostics was conducted, where the collinearity statistics Tolerance and VIF were calculated for the variables in the model. Menard (1995) suggests that a Tolerance value less than 0.1 almost certainly indicates a serious collinearity problem, and Myers (1990) suggests that a VIF value greater than 10 is cause for concern. All the independent variables displayed Tolerance values over 0.4 and the VIF values ranged from 1.008 to 3.891. These results suggest that the presence of multicollinearity is not a major problem in this regression analysis.

We conducted additional statistical analyses including only the variable with the largest odds for a file *being* problematic, i.e., ISP violation. We derived a *Confusion Matrix* as shown in Table 11. Similarly to the classification performance previously shown for the whole model in Table 9, the *recall* when using only ISPV is still not high (19.7), but we also have to consider that ISPV *alone* performs considerably well compared to the model including all the code smells.

4.3 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 = 0.604, and all KMO values for individual items were > 0.5, which is above the acceptable limit according to Kaiser (1974). Bartlett's test of Sphericity $\chi^2(66) = 561.252$, $p < 0.001$, 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.

Table 12 shows the factor loadings after rotation. When observing the factors, we see that the code smells God method and God class are the closest in Factor 1, 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, it is natural that they appear together. Also, large classes often use many different variables, which increases the

Table 11 Confusion matrix with variables 'ISP violation' and 'problematic?'

Observed	Classified		
	'ISP violation'		Percentage correct
	0	1	
'Problematic?'			
0	226	14	94.2
1	49	12	19.7

Table 12 Factor loadings after rotation

	Component				
	1	2	3	4	5
GM	0.751				
GC	0.730				
TMP	0.687				
DUP	0.595				
FE	0.537				
SS		0.896			
ISPV		0.823			
DC			0.751		
CL			0.721		
IMP				0.823	
RB					0.822
MC					−0.548
Eigenvalues	2.442	1.768	1.305	1.073	1.033
% of variance	20.350	14.731	10.875	8.942	8.607

chances of the presence of Temporary variable used for several purposes. Feature envy is also present when there are complex methods that need many parameters from other classes. Factor 1 variables may, consequently, be considered to relate to the size of the code. 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). In Section 4.4, we discuss the findings related to ISP violation in the light of the qualitative evidence gathered during the maintenance work. Data class and Data clump are together in one factor (Factor 3). Implementation instead of interface seems to appear very seldom in our dataset and does not relate to any of the other code smells (Factor 4). Refused bequest and Misplaced class constitute the last factor (Factor 5), where Misplaced class has a negative loading. This indicates that Misplaced class tends to be negatively associated with this factor.[2]

4.4 Problems Related to ISP Violation

In the model reported in Section 4.2, the coefficient of the ISP Violation variable is significantly different from zero, and its presence is connected with a high likelihood of observing a problematic file.

This section provides further analysis on ISP violation. First, based on the qualitative data collected during the interviews and think-aloud sessions, a report on how this code smell caused different types of difficulties for developers during maintenance is presented (Section 4.4.1). Subsequently, two significant cases observed during the study are described, where the interaction of ISP violation with other

[2]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 (Dunteman 1989).

code characteristics had acute consequences for maintainability (Section 4.4.2). The process of selecting the observations for Sections 4.4.1 and 4.4.2 were based on an examination of the qualitative information, resulting in several in-depth descriptions of maintenance problems deemed to be related to ISP violation. Out of these observations, we selected those deemed to be most representative and illustrative for the nature of ISP violation-related problems among the software developers.

4.4.1 Maintenance Problems Related to ISP Violation

The analysis on the qualitative data exposed a relationship between ISP violation and some of the negative consequences entailed by *wide afferent coupling*. Note that the detection strategy for ISP violation (See Appendix A) uses several afferent coupling measures such as Class Interface Width (CIW), and Clients Of Class (COC) (Marinescu 2002).

An analysis of the data from the think-aloud sessions showed that when the developers *introduced defects* in files with wide afferent coupling (thus, displaying ISP violation), the consequences of these defects 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. As an illustration, the classes located in the files `Nuller.java`, `StudyDAO.java`, and `DB.java` were found to propagate erratic behavior across different parts of the system and were associated with three maintenance problems reported by two developers.

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. These observations mainly came from the daily interviews when developers mentioned certain files one day (e.g., "I am working with files X, Y"), and the next day they reported that the changes in those files demanded changes in more areas of the code and would introduce delays in the project. To further illustrate problems related to ISP violation, we will present two observations from think-aloud sessions and daily interviews suggesting that problems with the developers' *program comprehension* are also related to the presence of this code smell.

The first observation relates to the presence of crosscutting concerns. In System A, the domain entity "Person" was being used within the "User" context and also within the "Employee" context. As such, the management of privileges/access, and information/functionality related to employees constituted crosscutting concerns. The crosscutting concerns manifested in System A, with the entity "Person" (located in the file `PeopleDatabase.java`) being accessed by many segments of the system. According to developers, this made the identification of the relevant task context very difficult.

The second observation relates to the presence of inconsistent design (manifested in the file `StudySortBean.java`, 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*[3] 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 (2002) 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 only Data class, acquired the ISP violation smell.

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.

The complete set of maintenance problems related to ISP violation (See Appendix B) provides further details to support the analysis presented in this section.

4.4.2 Interactions Between ISP Violation 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.

The first case was observed in System B, and it was related to time-consuming changes and defects after initial changes. This type of problem was reported by all developers who updated System B. The developers who worked on System B wanted to replace two interfaces (located in the files `Persistable.java` and `PersistentObject.java`[4]) with one new interface to support a String ID type in order to complete Task 1. Recall that Task 1 consisted of modiying 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 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 (i.e., *ObjectStatementImpl*) displayed ISP violation, which resulted in

[3]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.

[4]These interfaces are part of the Persistence Framework. As explained previously, 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.

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.

The second case relates to the observation that all systems except for System B contained one single "Brain class" that "hoarded" most of the logic and functionality in those systems (They were located in the files `StudyDatabase.java`, `DB.java`, and `StudyDAO.java`). 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.

In general, the qualitative data seems to support the results from the regression model and suggests that ISP violation constitutes a promising indicator for identifying problematic areas in the code.

4.5 Analysis of Files Containing Data Clump

The only other statistically significant coefficient in the regression model was related to the code smell Data clump. Surprisingly, the odds ratio of the variable related to this code smell suggests that its presence is connected with a low likelihood of observing a problematic file. This is supported by the fact that of the total set of 14 files modified by developers during maintenance that contained Data clump, only one file was deemed as problematic. The file `StudySortBean.java`, located in System A (previously described in Section 4.4) contained Data clump but also violated the Interface Segregation Principle. As we explained earlier, it seems as if ISP violation was the main reason why the file was considered problematic. Most of the files containing Data clump also constituted Data classes, as described in the PCA analysis (Section 4.3) and are used as data containers in the systems.

After a systematic search through all the qualitative data related to the Data clump and, the possibly connected, Data classes, it was not possible to find evidence explaining why these code smells were connected with a lower rather than a higher likelihood of maintenance problems. The lack of qualitative evidence points a limitation of the study design, i.e., the strong focus on "maintenance problems" when

collecting the qualitative data. With this focus, it is possible that some evidence was missed on the potentially positive effects of some code smells. In addition, it was observed during the study that developers rarely talked about attributes at class level that they were fond of. Developers were found to be more prone to complain about classes rather than express positive comments.

The results of this study suggest that some code smells could actually constitute beneficial attributes in the code, and as such, they represent an interesting topic for further studies.

5 Discussion of Results

5.1 Comparison with Related Work

Within the current literature on code smells, no study has yet reported the potential negative effects of ISP violation in maintenance projects. Nevertheless, the results of this study can be related to results by Li and Shatnawi (2007), who reported that Shotgun Surgery was positively associated with defects. In our study, it was observed that many instances of ISP violation were accompanied by the Shotgun surgery code smell, which suggests that the detection strategies of these code smells may uncover related types of design shortcomings.

It was also observed that in some cases, interaction between ISP violation and other code smells (or other design properties), seemed to cause maintenance problems. We know of only one study (by Abbes et al. 2011) that reported on the interaction between code smells (i.e., between God class and God method) and none between ISP violation and other code smells. The effects of code smell combinations seems to be a topic that deserves more attention. The importance of studying the interactions between code smells is further supported by observations that, in some large classes, the maintenance problems may not be directly caused by the actual size of the class, but rather are a result of interaction effects across different code smells that happen to appear together in the same file (See our discussion in Section 4.4.2). Olbrich et al. (2010) reported that when normalized with respect to size, classes constituting God class or Brain class had less defects and demanded less effort. This means that not all God classes or Brain classes are harmful, but it would require additional indicators to determine which instances are harmful, and how harmful they could be. Schumacher et al. (2010) asserts that automatic detection of code smells (in particular, God class) followed by a second human review can be a feasible approach for detecting potentially problematic code. An option, based on the results from this work would be, to use the presence of ISP violation to discriminate between instances of God class that are and are not likely to be harmful.

It has to be acknowledged that a recall of 37 % not too high. This means that we should only have moderate expectations on the derived model to explain potential maintenance problems. This further emphasizes one main finding, that code smells do have limitations in identifying problematic code. Also, 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 the model proposed in this paper, Data clump is negatively associated with maintenance difficulties. This suggests that there are code smells that, at least in some contexts, may lead to a lower risk of maintenance problems. Li and Shatnawi (2007) reported that Data classes were not significantly associated with defects. This may be the same as what was found here, given that the code smells Data Class and Data clump have a tendency to appear in the same file as shown in the factor analysis.

5.2 Threats to Validity

The validity of the study is considered and presented from three perspectives:

Construct Validity The code smells were identified via detection tools to avoid subjective bias in their identification. Nevertheless, the lack of standard 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 to some extent differ from those used in this study.

Internal Validity It is possible that some developers were more open about problems than others, and some did not report all the maintenance problems they experienced. Another threat to validity is the fact that 'round' can influence the developers' perception on what constitutes a problem and what not. These threats are common whenever qualitative data of the type collected in our study (interviews and think-aloud sessions) is used. We conducted a post-hoc analysis and found no significant effects from the variables *Developer* or *Round*. The R^2 values of the original model remained very similar to a second model including these variables, and the significance levels for the variables *System*, *Churn*, and the code smells *ISP violation* and *Data clump*, remained statistically significant. Our usage of three independent collection methods, i.e., interviews, direct observation and think-aloud sessions for triangulation[5] purposes, 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 in teams or pair programming. This last characteristic can affect the applicability of the results in highly collaborative environments.

We do not claim our results fully represent long-term maintenance projects with large tasks, given the size of the tasks and the shorter maintenance period covered in our study. However, the tasks involved may resemble backlog items in a single sprint or iteration within the context of Agile development. To the best of our knowledge,

[5]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."

we do not know of other experimental studies of code smells on in-vivo maintenance tasks for more than 240 min. In this study, we could closely observe the maintenance process for a period up to four full-working weeks.

Finally, some code smells may be more important for other types of maintenance tasks (e.g., corrective tasks, which were not investigated in this study) and other software applications than those studied in our study. For instance, in Table 7, it is possible to observe that some code smells seldom occur in the systems (e.g., Use of implementation instead of interface and Misplaced class). For those code smells, it is unlikely that the regression model would report coefficients significantly different from zero, unless the effect size is very large. Consequently, we cannot exclude the importance of the other investigated code smells without significant coefficients in the regression model developed in our study. The position taken in this paper is that is better to include the variables with low occurrence (and despite the risks entailed by multicollinearity), because if the omitted variables are correlated with included variables this would bias the coefficients (see discussion in Clarke 2005).

5.3 Implications

Results from this study point to ISP violation as a code smell associated with a non-negligible set of maintenance problems.

We believe that our study is a useful step toward building a more detailed causal chain that addresses how code smells affect maintenance outcomes. Since severity levels of the maintenance problems were not considered, the effect size of the ISP violation is yet to be investigated in detail. However, the data from the qualitative analysis suggests that ISP violation does not only frequently lead to maintenance problems, but also that the maintenance problems caused by this code smell may have a substantial impact on maintenance.

Results from this study reveal the inherent complexity of code smell analysis, where the effect of interactions across code smells may have potentially severe consequences on maintenance. Consequently, we believe that further research should investigate code smell combinations besides the study of individual code smells, alongside the notion of *inter-smell relations* posed by Walter and Pietrzak (2005).

We also identified instances of files with zero code smells, which turned out to be extremely problematic due to other design limitations and their coupling with ISP violators. Consequently, we believe that dependency analysis should be incorporated as an important component in further studies on code smells.

From a practical perspective, our study could contribute to maintainability analyses in industry, where components displaying ISP violation could be given particular attention to determine their design quality. This study provides a description of types of maintenance problems that ISP violation may cause, and developers and architects can associate these descriptions with situations they face in their projects to make refactoring/restructuring decisions.

6 Conclusion and Future Work

This research aimed at assessing the capability of twelve code smells to explain maintenance problems. We found evidence that ISP violation constitutes a code smell

likely to be associated with problematic files during maintenance. We found that Data clump was the code smell associated with the lowest probability of problematic files.

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 foreseeing maintenance problems.

We recommend, based on our findings and experience with the current study design, that future studies should include code smell analyses that: (1) include measures indicating the severity of the maintenance problems, and, (2) focus on the interaction effect between code smells, between code smells and other design properties, and between code smells and program size.

Acknowledgements The author thanks Gunnar Bergersen for his support in selecting the developers of this study and Hans Christian Benestad for providing technical support in the planning stage of the study. Also, thanks to Bente Anda and Dag Sjøberg for finding the resources needed to conduct this study and for insightful discussions. Thanks to Erik Arisholm for sharing his expertise during the analysis of the data. Finally, special thanks to Magne Jørgensen for his guidance and discussions that led to the paper. This work was partly funded by Simula Research Laboratory and the Research Council of Norway through the projects AGILE, grant no. 179851/I40, and TeamIT, grant no. 193236/I40.

Appendix A

Table 13 Code smells analyzed (part 1) and detection strategies (Marinescu 2002)

Code smells	Detection strategy	Metrics
Data class	WOC < 33 and (NOPA > 5 or NAM > 5)	Weight Of Class (WOC) Number Of Public Attributes (NOPA) Number of Accessor Methods (NAM)
Feature envy	AID > 4 and AID top 10 % and ALD < 3 and NIC < 3	Access of Import Data (AID) Access of Local Data (ALD) Number of Import Classes (NIC)
God class	AOFD top 20 % and AOFD > 4 and WMPC1 > 20 and TCC < 33	Access Of Foreign Data (AOFD) Weighted Methods Per Class 1 (WMPC1) Tight Class Cohesion (TCC)
God method	(LOC top 20 % except LOC < 70) and (NOP > 4 or NOLV > 4) and MNOB > 4	Lines Of Code (LOC) Number Of Parameters (NOP) Number Of Local Variables (NOLV) Max Number Of Branches (MNOB)
ISP violation	(CIW top 20 % except CIW < 10) and AUF < 50 and COC > 3	Class Interface Width (CIW) Average Use of Interface (AUF) Clients Of Class (COC)
Misplaced class	CL < 0.33 and NOED top 25 % and NOED > 6 and DD < 3	Number Of External Dependencies (NOED) Class Locality (CL) Dependency Dispersion (DD)
Refused bequest	AIUR < 1	Average Inheritance Usage Ratio (AIUR)
Shotgun surgery	CM top 20 % and CM > 10 and ChC > 5	Changing Methods (CM) Changing Classes (ChC)

Table 14 Code smells analyzed (part 2) and potential detection heuristics

Code smells	Detection heuristic
Data clump	*Abstract semantic graph* (Mamas and Kontogiannis 2000) 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* (Fischer et al. 2007) can be analyzed to detect conditional statements, and this information can be combined with *clone detection techniques* (e.g., Baxter et al. 1998)
Temporary variable is used for several purposes	Analysis of *abstract semantic graph* can be combined with *semantic analysis* (e.g., Landauer et al. 1998) to determine the location where temporal variables are defined and determine differences in their context of usage
Use interface instead of implementation	*Abstract semantic graph* can be analyzed to detect castings to implementation classes

Appendix B

Table 15 Problems report for file(s) containing ISP violation

ID	Summary	Type	Reason	Dev	Sys	File
3	Finding task context is difficult for *"Person"* and *"Publication"*, specially publication since domain is localized but code is spread	Understanding	Design inconsistency	2	B	Person
12	Variable "Search" used in different contexts, make it difficult to understand	Understanding	Design inconsistency	3	C	DB
15	Bug due to temporal variables repeatedely used for different purposes	Defects from side effects	Inconsistent variables	3	C	DB
20	Difficulties understanding the caching system	Understanding	Pervasiveness	2	B	Person Simula
28	Defects introduced after modifying function create new study (last modified/created by info lost during int-string conversion)	Defects from side effects	Large classes	2	B	Person
28	Defects introduced after modifying function create new study (last modified/created by info lost during int-string conversion)	Defects from side effects	Large classes	2	B	Simula
28	Defects introduced after modifying function create new study (last modified/created by info lost during int-string conversion)	Defects from side effects	Large classes	2	B	ObjectStatementImpl

Table 15 (continued)

ID	Summary	Type	Reason	Dev	Sys	File
42	Complaints about the size of StudyDAO	Understanding	Large classes	3	D	StudyDAO
43	Debugging problems during task 1, needed to change strategy several times	Modifying	Lack of flexibility flexibility	3	D	StudyDAO
45	Difficulties changing the sql queries to adapt to WS	Modifying	Data dependencies	1	D	StudyDAO
66	Difficulties finding the logic for displaying studies	Understanding	Logic spread	4	D	StudyDAO
70	Changes in DB.java triggered a series of errors in jsp files, which are difficult to track	Defects from side effects	Large classes	5	C	DB
75	Difficulties understanding the data retrieval (queries) for searching studies	Understanding	Logic spread	4	D	StudyDAO
77	Defects introduced after task 1	Defects from side effects	Large classes	6	A	StudyDatabase
78	Difficulties due to copy-paste error (1 h ca) from ResultSet variable which was wrong	Defects from side effects	Inconsistent variables	5	C	DB
80	Difficulties understanding business logic due to inconsistencies in the use of methods	Understanding	Design inconsistency	5	C	DB
82	Difficulties while implementing title based search and free text search (considered very difficult)	Modifying	Data dependencies	4	D	StudyDAO
88	Duplicated code in DB statements and connectors	Understanding	Design inconsistency	5	C	DB
91	Problems with change spread in DB (many methods)	Modifying	Large classes	5	C	DB
97	Difficulties due to side-effects from MT3 (isReallyNull) which required adjustment in order to cope with changes in StudyReport	Defects from side effects	Afferent coupling	4	D	Nuller
98	Some corrections were needed due to side-effects from tasks	Defects from side effects	Afferent coupling	4	D	StudyDAO

Table 15 (continued)

ID	Summary	Type	Reason	Dev	Sys	File
113	Side-effect from task 1	Defects from side effects	Afferent coupling	4	C	DB
118	Difficulties understanding and using the framework for displaying data	Understanding	Pervasiveness	6	B	Table
137	Side effects (bugs) from code reuse in duplicated logic	Defects from side effects	Inconsistent variables	1	A	StudyDatabase
35	Programmer doesn't understand why StudySortBean is used for study responsibles	Understanding	Semantic inconsistency	1	A	StudySortBean
36	Considered potentially difficult by the developer	Understanding	Design inconsistency	1	A	StudySortBean
47	Difficulties generating the url, related to difficulties with csv strings of ids	Modifying	Implementation	3	D	StudySDTO
50	WebConstants and WebKeys had hard-coded class names and were difficult to refactor, also Frontcontroller_URL had hardcoded paths	Modifying	Implementation	3	D	WebConstants, WebKeysStudy
65	Difficulties changing the hashmap so used caching instead	Modifying	Implementation	1	D	StudyDAO
127	Problems with space characters ("code was strange")	Understanding	Implementation	4	C	DB
26	Defect due to entityKey	Defects from side effects	Internal complexity + coupled to wide interface	2	B	ObjectStatementImpl
27	DB driver problems, and this forced the programmer to do hard casting from long to int	Modifying	Limited design + coupled to wide interface	2	B	ObjectStatementImpl

Appendix C

Table 16 Measurement values and summary statistics of code smells

Code smell	System A		System B		System C		System D	
	Tot.	Task	Tot.	Task	Tot.	Task	Tot.	Task
Data class (DC)	12	10	32	25	9	8	24	21
Data clump (CL)	8	6	2	1	3	3	8	5
Duplicated code in conditional branches (DUP)	1	1	4	4	2	2	2	2
Feature envy (FE)	37	37	34	32	17	17	25	23
God class (GC)	1	1	5	5	3	3	2	2
God method (GM)	4	4	14	12	3	3	5	5
ISP violation (ISPV)	7	6	8	8	1	1	11	11
Misplaced class (MC)	0	0	2	2	0	0	2	2
Refused bequest (RB)	17	15	8	8	0	0	1	1
Shotgun surgery (SS)	7	7	17	15	0	0	13	12
Temp. variable for several purposes (TMP)	12	12	31	27	6	6	4	4
Implementation instead of interface (IMP)	5	4	4	4	0	0	0	0
Total	111	103	161	143	44	43	97	88

Means: Total smells = 103.25, Task related smells = 94.25
Standard Deviation: Total smells = 48.11, Task related smells = 41.31

In the columns, 'Tot' indicates Total number of code smells in the system, and 'Task' indicates the number of code smells within the files that were modified/inspected by at least one developer during maintenance (i.e., 'Task related smells')

References

Abbes M, Khomh F, Guéhéneuc YG, Antoniol G (2011) An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: European conf. softw. maint. and reeng., pp 181–190

Alikacem EH, Sahraoui HA (2009) A metric extraction framework based on a high-level description language. In: Int'l conf. on source code analysis and manipulation (SCAM), pp 159–167

Anda BCD, Sjøberg DIK, Mockus A (2009) Variability and reproducibility in software engineering: a study of four companies that developed the same system. IEEE Trans Softw Eng 35(3):407–429

Baxter ID, Yahin A, Moura L, Sant'Anna M, Bier L (1998) Clone detection using abstract syntax trees. In: Int'l conf. softw. maint., pp 368–377

Bergersen GR, Gustafsson JE (2011) Programming skill, knowledge, and working memory among professional software developers from an investment theory perspective. J Individ Differences 32(4):201–209

Borland (2012) Borland Together. http://www.borland.com/us/products/together. Accessed 10 May 2012

Brown W, Malveau R, McCormick S, Tom Mowbray (1998) AntiPatterns: refactoring software, architectures, and projects in crisis. John Wiley & Sons, Inc

Clarke K (2005) The phantom menace: omitted variable bias in econometric research. Confl Manage Peace Sci 22(4):341–352

Coad P, Yourdon E (1991) Object-oriented design. Prentice Hall, London

D'Ambros M, Bacchelli A, Lanza M (2010) On the impact of design flaws on software defects. In: Int'l conf. quality softw., pp 23–31

Deligiannis I, Shepperd M, Roumeliotis M, Stamelos I (2003) An empirical investigation of an object-oriented design heuristic for maintainability. J Syst Softw 65(2):127–139
Deligiannis I, Stamelos I, Angelis L, Roumeliotis M, Shepperd M (2004) A controlled experiment investigation of an object-oriented design heuristic for maintainability. J Syst Softw 72(2):129–143
Dunteman GE (1989) Principal components analysis. Sage university paper series on quantitative applications in the social sciences. Sage, Newbury Park, CA
Edberg D, Olfman L (2001) Organizational learning through the process of enhancing information systems. In: Int'l conf. on system sciences, pp 1–10
Edgewall-Software (2012) Trac. http://trac.edgewall.org. Accessed 10 May 2012
Field A (2009) Discovering statistics using SPSS, 3rd edn. SAGE Publications
Fischer G, Lusiardi J, Wolff von Gudenberg J (2007) Abstract syntax trees—and their role in model driven software development. In: Int'l conf. on advances in softw. eng., pp 38–38
Fokaefs M, Tsantalis N, Chatzigeorgiou A (2007) JDeodorant: identification and removal of feature envy bad smells. In: Int'l conf. softw. maint., pp 519–520
Fowler M (1999) Refactoring: improving the design of existing code. Addison-Wesley, Reading, MA
Gamma E, Helm R, Johnson R, Vlissides J (1994) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading, MA
Intooitus (2012) InCode. http://www.intooitus.com/inCode.html. Accessed 10 May 2012
Juergens E, Deissenboeck F, Hummel B, Wagner S (2009) Do code clones matter? In: Int'l conf. softw. eng., pp 485–495
Kaiser H (1974) An index of factorial simplicity. Psychometrika 39(1):31–36
Khomh F, Di Penta M, Guéhéneuc YG (2009) An exploratory study of the impact of code smells on software change-proneness. In: Working conf. reverse eng., pp 75–84
Kiefer C, Bernstein A, Tappolet J (2007) Mining software repositories with iSPAROL and a software evolution ontology. In: Int'l workshop on mining softw. Repositories, pp 1–10
Kim M, Sazawal V, Notkin D, Murphy GC (2005) An empirical study of code clone genealogies. In: European softw. eng. conf. and ACM SIGSOFT symposium on foundations of softw. eng., pp 187–196
Landauer TK, Foltz PW, Laham D (1998) An introduction to latent semantic analysis. Discourse Process 25(2–3):259–284
Lanza M, Marinescu R (2005) Object-oriented metrics in practice. Springer
Larman C (2004) Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development, 3rd edn. Prentice Hall
Layman LM, Williams LA, St Amant R (2008) MimEc. In: Int'l workshop on coperative and human aspects of softw. eng., CHASE '08, pp 73–76
Li W, Shatnawi R (2007) An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J Syst Softw 80(7):1120–1128
Lozano A, Wermelinger M (2008) Assessing the effect of clones on changeability. In: Int'l conf. softw. maint., pp 227–236
Mamas E, Kontogiannis K (2000) Towards portable source code representations using XML. In: Working conf. reverse eng., pp 172–182
Mäntylä MV (2005) An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In: Int'l conf. softw. eng., pp 277–286
Mäntylä MV, Lassenius C (2006) Subjective evaluation of software evolvability using code smells: an empirical study. Empir Softw Eng 11(3):395–431
Mäntylä MV, Vanhanen J, Lassenius C (2004) Bad smells -humans as code critics. In: Int'l conf. softw. maint., pp 399–408
Marinescu R (2002) Measurement and quality in object oriented design. Doctoral thesis, "Politehnica" University of Timisoara
Marinescu R (2005) Measurement and quality in object-oriented design. In: Int'l conf. softw. maint., pp 701–704
Marinescu R, Ratiu D (2004) Quantifying the quality of object-oriented design: the factor-strategy model. In: Working conf. reverse eng., pp 192–201
Martin RC (2002) Agile software development, principles, patterns and practice. Prentice Hall
Menard S (1995) Applied logistic regression analysis. SAGE Publications
Moha N (2007) Detection and correction of design defects in object-oriented designs. In: ACM SIGPLAN conf. on object-oriented programming, systems, languages, and applications, pp 949–950
Moha N, Guéhéneuc YG, Leduc P (2006) Automatic generation of detection algorithms for design defects. In: IEEE/ACM int'l conf. on automated softw. eng., pp 297–300

Moha N, Guéhéneuc YG, Le Meur AF, Duchien L (2008) A domain analysis to specify design defects and generate detection algorithms. In: Fundamental approaches to softw. eng., pp 276–291
Moha N, Guéhéneuc YG, Duchien L, Le Meur AF (2010) DECOR: a method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36(1):20–36
Monden A, Nakae D, Kamiya T, Sato S, Matsumoto K (2002) Software quality analysis by code clones in industrial legacy software. In: IEEE symposium on softw. metr., pp 87–94
Myers R (1990) Classical and modern regression with applications. PSW-Kent Publishing Company, Boston, MA
Olbrich SM, Cruzes DS, Sjøberg DIK (2010) Are all code smells harmful? A study of God classes and brain classes in the evolution of three open source systems. In: Int'l conf. softw. maint., pp 1–10
Oracle (2012) My Sql http://www.mysql.com. Accessed 10 May 2012
Plone Foundation (2012) Plone CMS: open source content management. http://plone.org. Accessed 10 May 2012
Rahman F, Bird C, Devanbu P (2010) Clones: what is that smell? In: Working conf. on mining softw. Repositories, pp 72–81
Rajlich VT, Gosavi P (2004) Incremental change in object-oriented programming. IEEE Softw 21(4):62–69
Rao AA, Reddy KN (2008) Detecting bad smells in object oriented design using design change propagation probability matrix. In: Int'l multiconf. of eng. and computer scientists, pp 1001–1007
Riel AJ (1996) Object-oriented design heuristics, 1st edn. Addison-Wesley, Boston, MA
Schumacher J, Zazworka N, Shull F, Seaman C, Shaw M (2010) Building empirical support for automated code smell detection. In: Int'l symposium on empirical softw. eng. and measurement, ESEM '10, pp 1–8
Strauss A, Corbin J (1998) Basics of qualitative research: techniques and procedures for developing grounded theory. SAGE Publications
The Apache Software Foundation (2012a) Apache Subversion. http://subversion.apache.org. Accessed 10 May 2012
The Apache Software Foundation (2012b) Apache Tomcat. http://tomcat.apache.org. Accessed 10 May 2012
TMate-Sofware (2010) SVNKit - Subversioning for Java. http://svnkit.com. Accessed 10 May 2012
Tsantalis N, Chaikalis T, Chatzigeorgiou A (2008) JDeodorant: identification and removal of type-checking bad smells. In: European conf. softw. maint. and reeng., pp 329–331
Van Emden E, Moonen L (2001) Java quality assurance by detecting code smells. In: Working conf. reverse eng., pp 97–106
Vetro A, Morisio M, Torchiano M (2011) An empirical validation of FindBugs issues related to defects. In: Conf. on evaluation & assessment in softw. eng. (EASE), pp 144–153
Walter B, Pietrzak B (2005) Multi-criteria detection of bad smells in code with UTA method 2 data sources for smell detection. In: Extreme programming and agile processes in software engineering (XP). Springer, Berlin/Heidelberg, pp 154–161
Yamashita A (2012a) Assessing the capability of code smells to support software maintainability assessments: empirical inquiry and methodological approach. Doctoral thesis, University of Oslo
Yamashita A (2012b) Measuring the outcomes of a maintenance project: technical details and protocols (Report no. 2012-11). Tech. Rep., Simula Research Laboratory, Oslo
Yin R (2002) Case study research: design and methods (applied social research methods). SAGE
ZD Soft (2012) ZD soft screen recorder. http://www.zdsoft.com. Accessed 10 May 2012
Zhang M, Hall T, Baddoo N (2011) Code bad smells: a review of current knowledge. J Softw Maint Evol: Res Pract 23(3):179–202

Aiko Yamashita received a BSc. degree from the Costa Rica Institute of Technology in 2004 and a MSc. degree from Göteborgs University in 2007. In 2012, she finalized her doctoral degree at Simula Research Laboratory, and the Department of Informatics, University of Oslo. She has worked as a software engineer and consultant in Costa Rica, USA, Sweden and Norway within diverse organizations. Her research interests include empirical software engineering, software quality, software maintenance/evolution, source code analysis, psychology of programming, human-computer interaction, and agile methods.