Information and Software Technology 55 (2013) 2223–2242

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

To what extent can maintenance problems be predicted by code smell detection? – An empirical study

Aiko Yamashita [a,b,*], Leon Moonen [a]

[a] Simula Research Laboratory, P.O. Box 134, Lysaker, Norway
[b] Dept. of Informatics, University of Oslo, Oslo, Norway

ARTICLE INFO

Article history:
Received 17 July 2012
Received in revised form 5 August 2013
Accepted 10 August 2013
Available online 23 August 2013

Keywords:
Code smells
Maintainability
Empirical study

ABSTRACT

Context: Code smells are indicators of poor coding and design choices that can cause problems during software maintenance and evolution.
Objective: This study is aimed at a detailed investigation to which extent problems in maintenance projects can be predicted by the detection of currently known code smells.
Method: A multiple case study was conducted, in which the problems faced by six developers working on four different Java systems were registered on a daily basis, for a period up to four weeks. Where applicable, the files associated to the problems were registered. Code smells were detected in the pre-maintenance version of the systems, using the tools Borland Together and InCode. In-depth examination of quantitative and qualitative data was conducted to determine if the observed problems could be explained by the detected smells.
Results: From the total set of problems, roughly 30% percent were related to files containing code smells. In addition, interaction effects were observed amongst code smells, and between code smells and other code characteristics, and these effects led to severe problems during maintenance. Code smell interactions were observed between collocated smells (i.e., in the same file), and between coupled smells (i.e., spread over multiple files that were coupled).
Conclusions: The role of code smells on the overall system maintainability is relatively minor, thus complementary approaches are needed to achieve more comprehensive assessments of maintainability. Moreover, to improve the *explanatory power* of code smells, interaction effects amongst collocated smells and coupled smells should be taken into account during analysis.

1. Introduction

Significant effort and cost in software projects is allocated to maintenance [1–5], thus assessing the maintainability of a system is of vital importance. In the last decade, code smells have become an established concept for patterns or aspects of software design that may cause problems for further development and maintenance of these systems [6]. Code smell analysis allows people to integrate both assessment and improvement into the software evolution process itself.

Code smells are indicators that the code quality is not as good as it could have been, which can cause problems for developers during maintenance [7]. Code smells signal poor coding and design choices that degrade code quality aspects such as understandability and changeability, and can lead to the introduction of faults.

Beck and Fowler [7] provide a set of informal descriptions for 22 smells and associate them with different refactoring strategies that can be applied to improve software design. As such, code smell analysis opens up the possibility for integration of both assessment and improvement in the software maintenance process. Several tools are currently available for the automated detection of code smells, including commercial tools such as Borland Together[1] and InCode,[2] and academic tools such as JDeodorant [8,9] and iSPARQL [10].

Nevertheless, it is important for evaluations based on code smells, to understand better *how* these code characteristics cause *problems* during maintenance. Previous studies have investigated the relations between individual code smells and different maintenance outcomes such as effort, change size and defects; yet no study has investigated in detail, *how* and *which types* of *problems* code smells cause to developers during maintenance.

* Corresponding author at: Simula Research Laboratory, P.O. Box 134, Lysaker, Norway. Tel.: +47 47451242; fax: +47 67828201.
E-mail addresses: aiko@simula.no (A. Yamashita), leon.moonen@computer.org (L. Moonen).

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

0950-5849/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2013.08.002

In the context of this study, we define a *maintenance problem* as "*any struggle, hindrance, or challenge that was encountered by the developers while they performed their assigned tasks, which resulted in delays or in the introduction of defects during the maintenance project*." The *scope* of the maintenance problem is within programmatic activities (e.g., the ones described by Rajlich and Gosavi in [11] such as concept extraction/location, impact analysis, actualization, incorporation, change propagation, and other additional activities such as unit testing, debugging and configuration).

The study of maintenance problems is important, because problems can reflect and potentially explain different maintenance outcomes such as performance, product quality and perhaps even developers' motivational levels. The study of maintenance problems can provide important information for: (1) better understanding the relative impact of different (product- and process related) factors on maintainability and ultimately (2) building more detailed causal models of maintainability. If we have a better understanding the nature of potential maintenance problems that code smells can cause, we can make better-informed plans for code improvement.

This paper empirically investigates how much of the problems in a 'typical' web-application maintenance project can be explained by the presence of code smells. We report on a multiple case study in which the problems and challenges faced by six developers working on four different Java systems were registered on a daily basis, for a period up to four weeks. Observational notes and interview transcripts were used in order to identify and register the problems, and where applicable, the Java files associated to the problems were registered. The record of maintenance problems was examined and categorized into non-source code related and source code-related. Twelve different code smells were detected in the systems via Borland Together and InCode. In-depth examination followed in order to determine if the underlying cause(s) of the maintenance problems could be traced back to the presence of code smells in the associated files. When no code smells were present in the problematic code, we tried to identify any particular design characteristic that could explain the maintenance problem.

The remainder of this paper is structured as follows: Section 2 presents the theoretical background of this study. Section 3 presents the case study. Section 4 presents the results of the study. Section 5 discusses the results. Section 6 concludes and presents plans for future work.

2. Theoretical background and related work

2.1. Code smells

In [7], Beck and Fowler provided a set of informal descriptions for 22 code smells and associated them with different refactoring strategies that can be applied to improve software design. Code smells are characteristics that indicate degraded code qualities, such as comprehensibility and modifiability. As a result, code that exhibits code smells can be more difficult to maintain, which can lead to the introduction of faults.

Code smells have become an established concept for patterns or aspects of software design that may cause problems for further development and maintenance of the system [7,6]. They are also closely related to OO design principles, heuristics and patterns. Instances of OO design principles and heuristics can be found in the work by Riel [12] and Coad and Yourdon [13], seminal work on design patterns (and anti-patterns) can be found in [14–16], and in [17], Martin elaborates on a set of design principles advocated by the Agile community.

2.2. State of the art in code smell research

There has been a growing interest in the topic of code smells within the software engineering community after the publication of Fowler's refactoring book [7]. Van Emden and Moonen [18] provided the first formalization of code smells and described a tool for analyzing Java programs, while as Mäntylä [19] and Wake [20] proposed two initial taxonomies for code smells.

Two main approaches exist for the detection of code smells: Manual and Automated. The manual approach typically involves a subjective assessment, and the automated methods involve the use of source code analysis techniques to compute metrics or analyze properties. Travassos et al. [21] proposed a process based on manual detection, to identify code smells for quality evaluations. In [22,23] Mäntylä et al. report on an empirical study of subjective detection of code smells and compare it with automated metrics-based detection. They found that results from manual detection were not uniform between experienced developers and novices (e.g., experienced developers reported more complex smells). In addition, Mäntylä et al. found that developers with less experience with the modules reported more code smells than developers familiar with the modules.

Finally, when comparing subjective detection with automated, they found that developers' evaluations of complex code smells did not correlate with the results of the metrics detection. They conclude that subjective evaluations and metrics based detection should be used in combination. Mäntylä also reports on a experiment for evaluating subjective evaluation for code smells detection and refactoring decision [24]. He observed the highest inter-rater agreements between evaluators for simple code smells, but when the subjects were asked to make refactoring decisions, low agreement was observed.

Most of the current detection approaches for code smells are automated, and examples of such work can be found in [25–32]. Work on automated detection of code smells been used in commercial tools such as *Borland Together* and *InCode* and academic tools such as JDeodorant [8,9] and iSPARQL [10]. Zhang et al. [33] conducted a systematic literature review to describe the state of art in research pertaining code smells and refactoring. They covered papers published by IEEE and six leading software engineering journals from 2000 to June 2009. They found that very few studies report on empirical studies involving effects of code smells, and most studies focus on developing tools and methods for supporting automatic detection of code smells. Previous studies have investigated the effects of individual code smells on different maintainability related aspects, such as *defects* [34–38], *effort* [39–42] and *changes* [43–45].

Instead of first detecting bad smells in code that can then in turn be removed by applying the associated refactorings, some researchers have focused on alternative approaches for detecting refactoring opportunities. These approaches follow a more direct approach and try to immediately identify if a given refactoring can be applied using a variety of program analysis techniques and source code metrics. The approaches typically target a single refactoring, such as extract method [46], move method [47], pull up method [48], extract class [49,50], and form template method [51], the introduction of polymorphism [52], or a class of related refactorings, such as the potential for generalization [53] by means of clone detection. By applying the detected refactoring, the code will be improved, and any associated code smells may be removed as a side effect. These approaches are generally supported by prototype tools that can detect specific refactoring opportunities in the context of the particular study. Although such tools push the state of the art on a particular refactoring, they do not support the type of wide-spectrum code smell analysis that is needed to analyze the relation between code smells and maintainability

problems in an industrial context, as is done in the study presented here. Moreover, in the context of this study, we argue that the nature of maintainability problems can be better investigated and understood by means of code smells, aimed indicating poor coding and design choices, than by means of possible code transformations that aim to alleviate those poor choices.

D'Ambros et al. [37] analyzed code in the open source systems Lucene, Maven, Mina, CDT, PDE, UI, Equinox for existence of code smells described in [7], and found that neither *Feature Envy* nor *Shotgun Surgery* was consistently correlated with defects across systems. Juergens et al. [36] analyzed the proportion of inconsistently maintained *Duplicated Code* (i.e., divergent changes in code fragments that were once clones) in relation to the total set of duplicates in C#, Java and Cobol systems, and found (with the exception of Cobol) that 18% of them were positively associated to faults. Li et al. [35] investigated the relationship between six code smells and class error probability in an industrial-strength system, and found that *Shotgun Surgery* was positively associated with software faults. Monden et al. [34] performed an analysis of a Cobol legacy system and concluded that cloned modules were more reliable, but demanded more effort than non-cloned modules. Rahman et al. [38] conducted a descriptive analysis and non-parametric hypothesis testing of source code and bug tracker in the systems Apache httpd, Nautilus, Evolution and Gimp. They found that the majority of defective code is not significantly associated with clones (80% of defective code at system level contained zero clones), clones may be less prone to defects than non-cloned code, and clones that repeat less across the system are more error prone than more repetitive clones.

Abbes et al. [42] conducted an experiment in which 24 students and professionals were asked questions about the code in the open source systems YAMM, JVerFileSystem, AURA, GanttProject, JFreeChart and Xerces. They concluded that classes and methods exhibiting *God Class* and *God Method* smells in isolation had no effect on effort or quality of responses, but when these smells appeared together, they led to a statistically significant increase in required effort and a statistically significant decrease in percentage of correct answers. Deligiannis et al. [39] conducted an observational study where 4 participants evaluated two systems, one compliant and one non-compliant to the principle of avoiding *God Class*. Their main conclusion was that familiarity with the application domain played an important role when judging the negative effects of god class. They also conducted a controlled experiment [40] with 22 under-graduate students as participants, and could corroborate their initial findings that a design without a *God Class* resulted in better completeness, correctness and consistency than of the design that did contain a *God Class*. Lozano et al. [41] reported that existence of duplicated code increases maintenance effort on cloned methods. However, they were unable to identify characteristics that systematically revealed a significant relation between cloning and maintenance effort increase.

Khomh et al. [44] analyzed the source code of the Eclipse IDE and found that, in general, classes that exhibited the *Data Class* code smell, were changed more often than classes that did not exhibit this code smell. Kim et al. [43] reported on the analysis of two medium-sized open source libraries (Carol and dnsjava) and concluded that 36% of the total amount of code that had been duplicated changed consistently (i.e., they remained as identical duplicates via simultaneous updates), while the remaining evolved independently. Olbrich et al. [45] reported an experiment involving the analysis of three open source systems and found that classes exhibiting *God Class* or *Brain Class* smells were changed less frequently and had fewer defects than other classes when normalized with respect to size.

Recently, two studies have been published that have investigated the lifespan of code smells during the evolution of software systems [54,55]. Independently of each other, both author groups found that code smells accumulate in systems over time; smells are usually introduced when the method in which they reside was initially added; smells are almost never removed, but when they are it is usually shortly after they were added, and generally not as a result of targeted refactoring but as a side effect of other changes. Peters and Zaidman conclude that developers may be aware, but are not concerned by the existence of code smells [55].

The current state of art indicates that not all code smells are equally harmful. Also, code smells are not harmful to the same extent over different contexts, indicating that their effects are potentially contingent on contextual variables or interaction effects. For example, Li and Shatnawi [35] found that the presence of Shotgun Surgery leads to defects. D'Ambros et al. [37], on the other hand, found no such connection between Shotgun Surgery and defects. Results from studies on Duplicated Code suggest that the effects of duplication depend on factors such as the programming language; e.g., the results from the COBOL system differed from that of the other types of systems in the study by Juergens et al. [36]. In order to understand better the interaction between code smells and different contextual factors, we perceive that more inductive research approaches are needed. Case study design is an example of an approach that could significantly contribute to the development of new *theories* from observations in relevant fields and contexts (i.e., *inductive research* [56]).

2.3. Studies on maintenance problems

Several studies have addressed maintenance related problems, as well as factors causing them. In [57], Kitchenham et al. propose a maintenance ontology and describe a set of factors affecting maintenance.

Lientz and Swanson [58] and Dekleva [59] describe maintenance problems from a managerial perspective. For instance, [58] reports user knowledge, programmer effectiveness, product quality, programmer time availability, machine requirements, and system reliability as the main sources of problems during maintenance. Similarly, in [59] maintenance problems are elicited from experienced software developers, and four major categories are identified: maintenance management, organizational environment, personnel factors, and system characteristics. Palvia et al. [60] elaborate on the initial set of problems in software maintenance reported by Lientz and Swanson, and assess the criticality of the factors based on an extensive survey involving software practitioners.

Chapin [61] provides a classification for types of software maintenance and evolution, and discusses how their characteristics can impact different aspects of the product and the business. Hall et al. [62] and Chen et al. [63] describe maintenance problems and the factors that potentially cause them from a Software Process Improvement perspective, and Reedy [64] proposes a catalogue of maintenance problems within software configuration management.

Karahasanovic et al. [65,66] report on a catalogue of program comprehension problems observed during an experiment designed to investigate whether following systematic strategy for program comprehension is unrealistic in larger programs. They used the taxonomy for program comprehension proposed by Mayrhauser and Vann [67] and classified problems at the level of general knowledge (which is independent of the specific software application that the programmers are trying to understand, such as cache memory), and at the level of software-specific knowledge (which represents their level of understanding of the software application, such as the use of memory pointers in C programming language). They also compared which problems were associated to which type of comprehension strategy.

Webster [68] proposes taxonomy of risks in software maintenance, which can also be seen as potential factors negatively affecting software maintenance. Webster refers to [4,69–71], when describing a set of code and design characteristics that represent a risk for the maintenance process. Examples of risks include: Antiquated system and technology, program code complexity, problems in understanding programming 'tricks', large number of lines of code affected by change request, and large number of principal software functions affected by the change.

2.4. Knowledge gap

As seen in Section 2.2, there is a substantial body of work that investigates if certain source code characteristics (i.e., a code smell) affect a given maintenance outcome (e.g., effort, changes, defects). However, when assessing software maintainability, it is not only important to determine if a code characteristic have or not an effect on a given maintenance outcome, but we also need to better understand *how* those characteristics affect the outcomes. We believe that one approach to get a step further in the development of causal models in software maintenance research is by closely studying the *difficulties* or *problems* that developers experience during maintenance.

The study of *maintenance problems* is relevant because these experiences reflect and potentially explain different outcomes of a maintenance project, such as performance and product quality. Our scope for *maintenance problems* is based on the notion of *Incremental Change* proposed by Rajlich and Gosavi [11] who describe a series of stages and activities within an *increment* in the evolution of Object Oriented systems. Rajlich and Gosavi mention two stages: *change design*, where developers conduct activities such as: concept extraction, concept location, and impact analysis; and *change implementation*, which involves activities such as: change incorporation and propagation (see [11] for a detailed description of each of the activities). Although not mentioned in [11], we believe that activities such as: change testing, debugging, and configuration activities constitute part of the incremental change process.

We believe that the *maintenance problems* illustrated and described within the identified literature (i.e., Section 2.3), focus mostly on *process* and *organizational* aspects of software maintenance, parting from a rather managerial perspective. They do not address comprehensively, the problems that can manifest during programming activities within an *incremental change* in the evolution of a software product. For instance, Karahasanovic et al. only focuses on comprehension-related problems, whiles comprehension only accounts for two of the activities in an increment (i.e., concept extraction, concept location). The work by Lientz and Swanson, Dekleva, Palvia, and Chen all focus on management or software process improvement perspectives. They cover the whole spectrum of a software maintenance process, and mention code base quality in a rather superficial manner. The work by Webster [68] can be considered the closest in their focus, to our study.

Thus, the state of art on maintenance problems does not provide the level of abstraction required to study the effects of software design/code on maintainability, and the current taxonomies seem too coarse-grained to address the actual "programming" activity embedded within the maintenance/evolution process. The investigation of maintenance problems at this level of detail may help to address the causal "step" in-between code smells (or any other code characteristics) and maintenance outcomes that needs to be investigated.

In addition, when investigating the effects of code smells on maintenance, our focus on *problems* corresponds better to the original descriptions of code smells given by [7], where smell definitions are more direct symptoms of *problematic maintenance* rather than change effort, change size, and defects (as the empirical studies reported in Section 2.1). For example, *effort* can be affected negatively by programmer's skill, or the task, and not necessarily by the low maintainability of the system. To the best of our knowledge, a thorough, systematic analysis of the connection between maintenance problems and code smells in practice has not been conducted before, and has been, long overdue.

In the present study, an in-depth analysis has been carried out to identify maintenance problems that can be related back to technical properties of the system (i.e., code smells, characteristics of the source code). The results of this research are based on realistic maintenance tasks (or change requests) carried out by six professional software engineers, each one working for a period up to four weeks. Detailed data-collection was performed via interviews, direct observation and think-aloud techniques. The qualitative observations in the field were supported by quantitative data, opening up the possibility to develop theories on code smells that can be tested and investigated in further studies. The design of our study is based on the assumption that by observing the whole range of maintenance problems in a prototypical project, we can achieve a better understanding of the role that code smells (i.e., automatically detectable smells) play in maintenance, which, in turn, helps to further the area of automatic maintainability assessments based on source code analysis.

Previously, we have looked at how well the *definitions* of code smells address of characteristics that were deemed important for maintainability by developers [72]. The study reported in [72] was based on the same industrial case as the current paper and the data that was analyzed consisted of expert-based maintainability assessments of four Java systems before they entered a maintenance project and a series of open interviews with the developers after they had conducted the maintenance tasks. Using a combination of coding techniques and called cross-case synthesis, we analyzed the transcripts of these interviews to compare each developer's perception on the maintainability of the systems, extracted the important maintainability aspects, and relate them back to definitions of code smells. As such, the paper in [72] investigates the conceptual relation between code smell definitions and aspects of maintainability. This is of interest because it helps to focus on smells that really matter for developers, and identifies smells for which detection strategies are yet not available but would be very beneficial. The current paper, in contrast, evaluates to what extend the total set of problems occurring during maintenance can be explained by the actual presence of code smells in the code under maintenance. The data used as evidence is much more comprehensive than in the previous work, and includes the measurements of the code smells, daily interviews, think-aloud sessions and observational logs. The open interviews that played the main role earlier were used as confirmatory (triangulation) evidence in this study.

3. Case study design

This section starts with describing the background of the maintenance project. Subsequently, it describes the data sources used as the basis of the analysis and the methods used in order to attain the research goals.

3.1. Context of the study

3.1.1. Systems under analysis

To conduct a longitudinal study of software development, the Simula's Software Engineering Department put out a tender in 2003 for the development of a new web-based information system to keep track of their empirical studies. Based on the bids, four

Table 1
Development costs from [73] and final size (LOC) of the systems.

System	A	B	C	D
Cost	€ 25,370	€ 51,860	€ 18,020	€ 61,070
Java	8205	26,679	4983	9960
JSP	2527	2018	4591	1572
Other files	371	1183	1241	1018
Total size	11,103	29,880	10,815	12,550

Norwegian consultancy companies were hired to independently develop a version of the system, all using the same requirements specification. The companies knew, and agreed that the work would be done as part of a research study. More details on the initial project can be found in [73].

The same four functionally equivalent systems are used in our current study. We will refer to them as System A, System B, System C and System D, respectively. The systems were primarily developed in Java and they all have similar three-layered architectures, but have large differences in their design and implementation. Their cost and size differed notably as well (Table 1). The main functionality of the systems consisted of keeping a record of the empirical studies and related information at Simula (e.g., the researcher responsible of the study, participants, data collected and publications resulting from the study). Another key element of functionality was to generate a graphical report on the types of studies conducted per year. The systems were all deployed over Simula Research Laboratories' Content Management System (CMS), which at that time was based on PHP and a relational database system. The systems had to connect to the database in the CMS in order to access data related to researchers at Simula as well as information on the publications.

3.1.2. Maintenance project

In 2008, Simula's CMS was replaced by a new platform called Plone,[3] and it was no longer possible to run the systems under this new platform. This gave the opportunity to set up a maintenance study, where the functional similarity of the systems enabled investigating the relation between design aspects and maintainability on cases with very similar contexts (e.g., identical tasks and programming language), but different designs and implementations. This study was conducted between September and December 2008 by outsourcing the project to six professional developers at a total cost of €50,000.

3.1.3. Subjects

To conduct this maintenance project, six professional developers were recruited from a pool of 65 participants to a separate study on developer skills by one of our colleagues at Simula [74]. These developers had no relation to the original developers of the four systems. They were selected based on having roughly similar development skills, their availability, English proficiency, and high motivation for participating in the maintenance project, for which they received their regular salary. Although the pool contained representatives from nine companies in eight European countries, the six selected developers were employees of two software companies in Eastern Europe (three from Czech Republic and three from Poland). Both the developers and their companies were aware from the start that this maintenance project was part of a scientific study. In the remainder of this paper we will use the terms *developer* and *maintainer* interchangeably to refer to these subjects that conducted the maintenance tasks. In the rare cases that we want to refer to the original developers that initially created the systems, we will use the phrase *original developers*.

3.1.4. Activities and tools

The study was conducted in each of the companies premises. Although the three developers in each site shared the same office space, they were discouraged to discuss the project while in it. The developers were given an overview of the project and a specification of each maintenance task. When needed, they would discuss the maintenance tasks with the researcher (first author) who was present at the site during the entire project duration. Daily interviews were held where the progress and the problems encountered were tracked. Acceptance tests were conducted once all tasks were completed, and individual open interviews were conducted where the developer was asked upon his/her opinion of the system. The daily interviews and wrap-up interviews were recorded for further analysis. Eclipse was used as the development tool, together with MySQL[4] and Apache Tomcat.[5] Defects were registered in Trac[6] (a system similar to Bugzilla), and Subversion or SVN[7] was used as the versioning system.

3.1.5. Observed cases

Each of the six developers individually conducted all three tasks on two systems. This was done to collect more observations for different types of analysis, and gave us a total of 12 cases, 3 observations per system. The assignment of developers to systems was random, with control for equal representation, learning effects (i.e. every system at least once in first round and at least once in second round) and maximizing contrast between the two cases handled by each developer.

3.2. Data collection activities

3.2.1. Data sources

Several sources of evidence were used in this study, as listed in Table 3. Some data sources were used as primary evidence, and others were used for triangulation purposes. Three qualitative data collection activities were performed in order to attain the primary sources of evidence: (1) daily interviews or progress meetings, (2) random think-aloud sessions and (3) logbook annotations. Source code was used as the fourth main source of evidence, and it was examined for the presence of code smells. The rest of the sources in Table 3 (i.e., transcripts of the open-ended interviews, task progress sheets, change size derived from SVN, and maintenance effort derived from Eclipse activity logs) were used for triangulation purposes. The audio files from both types of interviews were transcribed, annotated and summarized with Transana.[8]

3.2.2. Maintenance tasks

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. We believe that these tasks are representative of realistic maintenance tasks for several reasons: (1) the tasks were based on concrete needs in a real-life situation on a system that was actually in use: both the change of environment on which the systems were running that lead to the adaptive tasks, and the new functionality were requested by several users of the system, (2) the tasks had non-trivial impacts on the systems and developers were required to examine and modify substantial parts

[3] http://plone.org.

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

Table 2
Maintenance tasks.

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 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, as now a String type is used instead. 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 and using authentication mechanisms available on that time for Simula Web site. This maintenance task consisted of replacing the existing authentication by calling a web service provided for this purpose
3	Add new reporting functionality	This functionality provides options for configuring personalized reports, where the user can choose the type of information related to a study to be included in the report, define inclusion criteria based on people responsible for 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 must be stored in the systems' database and should only be editable by the owner of the report configuration

Table 3
Sources of evidence used in the study.

No.	Data sources	Data collection activity
1	Daily interviews	Individual progress meetings (20–30 min): were conducted daily, with each of the developers and the researcher present at the study to keep track of the progress, and register problems encountered during the project (ex. Dev: "It took me 3 h to understand this method...")
2	Think aloud sessions	Think-aloud sessions (ca. 30 min) were conducted in random points to observe the developers in their daily activities. During these sessions, the developers' screens were recorded, for later analysis and triangulation. There was one developer who felt uncomfortable vocalizing their thoughts, in which case this session was limited to recording the screen and taking notes based on the researcher's observations
3	Logbook	The researcher present at the study kept a study diary or logbook in a daily basis, to annotate in detail the observations from all different data collection activities, and also to annotate any important aspects of the study
4	Source code	The system's source code. (Note: The scope of the code smells is limited only to Java files, and consequently, jsp, sql files and other artefacts were not considered for measurement)
5	Open-ended interviews	Individual open-ended interviews (40–60 min): were held after the maintenance tasks were completed for each system, where the developer gave his/her opinion of the system(s) on which he/she had worked so far (e.g., how difficult was it to understand the systems?)
6	Task progress sheets	The developers filled in these sheets, where they compared estimations vs. actual time for each of the sub-tasks required for the maintenance
7	Subversion database	The repository database where the source code was kept (Subversion was used)
8	Eclipse activity logs	Developer's activities were logged by a plug-in called Mimec [75]. This plug-in logged all the actions performed on Eclipse at the GUI level, including filenames and elements of the Eclipse GUI

of the systems (around 70% for System B which was the largest, and higher percentages for the smaller systems). For example, task 1 required the developers to find and modify data access procedures for the major domain entities in the system, and (3) the time to complete all three tasks on one system took up to three weeks per developer, which is equivalent to a single sprint or iteration in the context of Agile development. Trivial tasks do not demand such an extensive period of effort. Moreover, as was mentioned before, we are unaware of any other in vivo studies of software maintenance that lasted longer than 240 min.

3.2.3. Identification of maintenance problems

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 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 ZD Soft's Screen Recorder.[9] 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).

An example of how problems were identified is given in Table 4, which presents an excerpt of the observations recorded during a think aloud session. In this example, the observations by the researcher and the literal statements given by the developer during a think aloud session led to the interpretation that the initial strategy of replacing several interfaces in order to complete maintenance task 1 was not feasible due to unmanageable error propagation. The developer spent up to 20 min trying to follow the initial strategy, but then decided to rollback and to follow an alternative strategy (forced casting) whenever it was required.

A logbook or study diary (See source 3 in Table 3) 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.

Although the identification of problems can be subjective to some degree, the connections between the observations and the incidences of problems in this study were deemed to be rather

[9] http://www.zdsoft.com.

Table 4
Excerpt from think aloud session.

Code	Statement or action by developer	Observation/interpretation
Goal	Change entities' ID type from Integer to String	This is part of the requirements in Task 1
Finding	"Persistence is not used consistently across the system, only few of them are actually implementing this interface so..."	Persistence[a] is referred as two interfaces for defining business entities, which are associated to a third-party persistence library, which is not used consistently in the system
Strategy	"I will remove this dependency, I will remove two methods from the interface (getId an setId) added for integer and string. This strategy forces me to check the type of the class but this is better than having multiple type forced castings throughout the code"	Developer decides to replace two methods of the Persistence interface (i.e., getId () an setId ()) which are using Integer and will replace them with methods with String parameters
Action	Engages in the process of changing id in interface PersonStatement.java	Developer engages in the initial strategy
Muttering	"Uh, updates? just look at all these compilation errors..."	Developer encounters compilation errors after replacing the methods in the interfaces
Action	Fix, refactor, correct errors	Starts correcting the errors
Strategy	"Ok... I need to implement two types of interfaces, one for each type of ID for the domain entities. I will make PersistentObjectInt.java for entities that use Integer IDs and PersistentObjectString.java for String IDs"	Change of strategy, decides to actually replace the interface instead of replacing the methods in the interface
Action	Fix more errors from Persistable.java	More compilation errors appear
Action	Continue changing interface of the entity classes into PersistentObjectInt and PersistentObjectString	Attempt to continue with the second strategy
Action	(After 20 min) Roll back the change	Developer realizes that the amount of error propagation is ummanegeable, so rollbacks the changes
Muttering	"Hmm... how to do this?"	Developer thinks of alternative options
Strategy	"Ok, I will just have to do forced casting for the cases when the entity has String ID"	Developer decides to use the least desirable alternative: forced type castings whenever is required

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

Table 5
Code smells automatically detected in the systems.

Code smell	Description
Data Class	Classes with fields and getters and setters not implementing any function in particular
Data Clumps	Clumps of data items that are always found together either within classes or between classes
Duplicated code in conditional branches	Same or similar code structure repeated within a the branches of a conditional statement
Feature Envy	A method that seems more interested in data from a class other than the one it is actually in. Fowler recommends putting a method in the class that contains most of the data the method needs
God (Large) Class	A class has the God Class smell if the class takes too many responsibilities relative to the classes with which it is coupled. The God Class centralizes the system functionality in one class, which contradicts the decomposition design principles
God (Long) Method	A class has the God Method bad smell if at least one of its methods is very large compared to the other methods in the same class. God Method centralizes the class functionality in one method
Misplaced Class	In "God Packages" it happens often that a class needs the classes from other packages more than those from its own package
Refused Bequest	Subclasses don't want or need everything they inherit
Shotgun Surgery	A change in a class results in the need to make a lot of little changes in several classes
Temporary variable for various purposes	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 faults
Use interface instead of implementation	Castings to implementation classes should ideally be avoided and an interface should be defined and implemented instead
Interface Segregation Principle (ISP) Violation	The dependency of one class to another one should depend on the smallest possible interface. Even if there are objects that require non-cohesive interfaces, clients should see abstract base classes that are cohesive. Clients should not be forced to depend on methods they do not use, since this creates coupling

direct. A second researcher was presented during the kickoff week in Czech Republic. This second researcher was present at several of the data collection activities (i.e., daily interviews and think-aloud sessions). This enabled to cross-validate the initial observations, and both researcher's observations (and respective notes) were used as basis to create and revise the think-aloud schema. In addition, a portion of the daily interviews transcripts were examined by the second researcher, in order to validate the interpretations of the researcher who registered the maintenance problems.

3.2.4. Identification of code smells

Instances of twelve different code smells were automatically identified in the source code of the four systems using the commercially available tools Borland Together and InCode. The selection of these particular tools was based on: (1) the fact that vendors would reveal how they detect the smells, (2) the tools implemented well-known detection strategies that were comprehensively described by Marinescu and Lanza ([76,6]), and (3) the choice of these tools would enable cross-comparison with other studies that have investigated code smells (e.g., previous work on code smells by Li and Shatnawi [35], who have used Together).

The choice for using commercial detection tools is also justifiable from the perspective of realism, in that this selection resembles the choice that would be made in an industrial setting, and from the perspective of replication, since their general availability makes replication of our work straightforward.

Table 5 presents the list of code smells that were detected in the systems, alongside their descriptions, taken from [7]. Note that the last smell in the table, i.e., Interface Segregation Principle (ISP) violation is not a smell, but a design principle violation, proposed by Martin in [17]. Also, note that *Duplicated code in conditional branches* is a "local version" of code duplication or *code cloning* that looks for duplicated code located across conditional branches. This

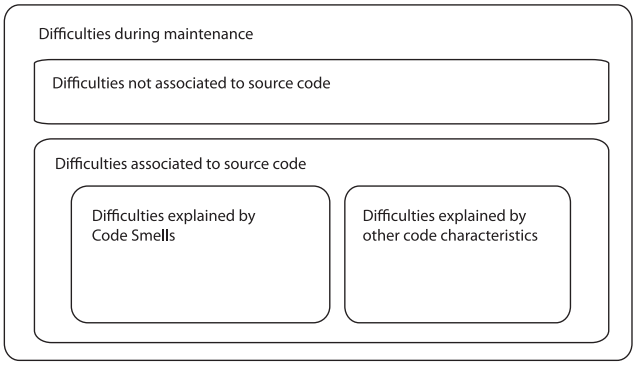

Fig. 1. Venn-diagram of maintenance problems.

local variant is of interest to us because this is a clear situation in which cloning can lead to mix-ups when maintaining the code. For code cloning in general on the other hand, there is empirical evidence that it is not always a bad practice but can be useful as strategy for creating code variants, such as drivers, that even though they start as clones, undergo little to no connected modifications during their lifetime [43,77,78]. As such, we leave the analysis of duplicated code in general as a topic that is beyond the scope of this study. Both Interface Segregation Principle (ISP) violation and Duplicated code in conditional branches can be detected using Borland Together.

3.3. Data analysis approach

In order to answer the research question, the problems identified were categorized into source code-related and non-source code related. The source code-related problems and JAVA files connected to them were examined in detail in order to determine if the underlying problem was explained or not by the presence of code smells.

The files that did not contain any detectable code smell were manually reviewed to analyze if they exhibited any characteristics or issues that could explain why they were associated to problems. This step is similar to doing code reviews for software inspection, where peers or experts review code for constructs that are known to lead to problems. In this case the task is easier because we already know that there is a problem associated with the file and we look for evidence that can explain this problem. Fig. 1 depicts the Venn-diagram of maintenance problems according to the categorization criteria we just described.

In addition, to determine whether multiple files contributed to maintenance problems instead of individual files, the notion of coupling was used as part of the analysis. The tools InCode and Stan4J[10] were used to identify such couplings. The notion of coupling used in these tools is the same as the one defined by Stevens et al. [79], where file (class or interface) A is coupled to B in any of the following cases:

(a) A has an attribute that refers to, or is of type B (i.e. A instantiates B).
(b) A calls on methods of an object B.
(c) A has a method that references B (via return type or parameter).
(d) A is a subclass of B (i.e. inheritance).
(e) A implements B (i.e. when B is an Interface and A implements the Interface).

[10] http://stan4j.com.

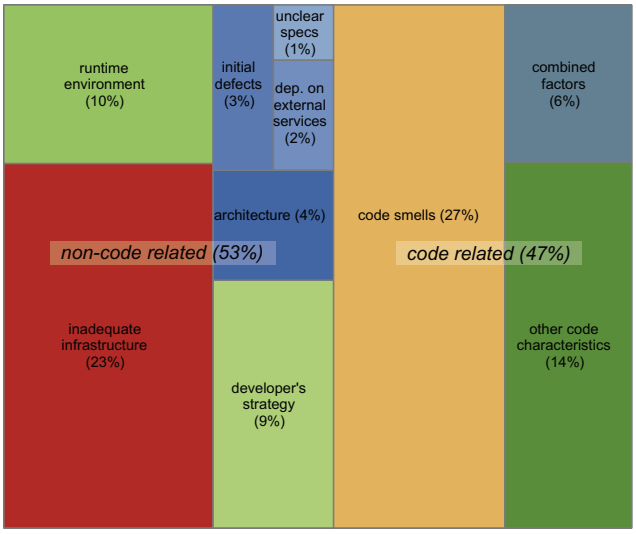

Fig. 2. Distribution of maintenance problems classified by their origins.

4. Results

This section first presents the distribution of the different classes of maintenance problems identified during the project. Second, the fraction of the problems related to code smells is described, alongside detailed examples. Third, we describe cases where other source code characteristics than code smells were the cause of the maintenance problems. Finally, details are given on cases where problems were due to a combination of factors (sometimes including code smells).

4.1. Overview of 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%) constituted problems not directly related to code such as: lack of adequate technical infrastructure, developer's coding habits, external services, runtime environment, and defects initially present in the system. The distribution of the observed problems classified by their origins is visualized in the treemap in Fig. 2. Appendix A presents an excerpt of the records where maintenance problems were registered. Each entry contains a summary of the problem, whether it was associated with Java code or not, files involved, developer who reported the problem, system, and the origin of the problem. 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.

Another way to classify the observed maintenance problems is to distinguish problem types based on tasks or activities. In this categorization, the majority of problems (91%) related to three clearly distinguishable situations: (a) defects introduced as result of changes (25%), (b) program comprehension and information searching (27%), and (c) time-consuming changes (39%). The remaining problems manifested at much lower scale, and include cumbersome configuration (6%) and debugging (3%). Table 6 describes the three major problem types in this activity related categorization in more detail, and Table 7 presents the distribution of the problems according to their associated activities and (high-level) origins (code related vs non-code related, and split up per system).

Table 6
Description of the three main types of problems 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 in several cases to rollback initial strategies for solving the tasks
Troublesome program comprehension and information searching	This problem type comprises three situations: One where developers struggle to get an overview of the system or to get high-level understanding of the system's behavior. The second situation relates to low-level understanding of the code, where developers become confused because they find inconsistent or contradictory evidence in different components of the system. The third case consists of time consuming or troublesome information or "task context" search (e.g., finding the place to perform the changes, finding the data needed to perform a task)
Time-consuming or costly changes	Time consuming or costly changes were associated with two situations: One where the nature of the task required making changes to a high number of components in the system, and the second one for tasks that were cognitively demanding due to the complexity of the problem to be solved, or due to intricate design, visualization or information distribution. While problems in the previous category all relate to information search and program comprehension, for problems in this category, we assume that the developer has already gathered the information and understands fairly well the code, but during the problem solving stage, the solution has to be reworked, or the solution is difficult to formulate

Table 7
Difficulties according to *source* and *type*.

	Sys.	Introd. defects	Program compreh.	Costly changes	Config.	Debug	Total
Not code related	A	7	6	2	4	0	19
	B	1	0	2	4	0	7
	C	5	2	17	0	5	29
	D	4	5	9	0	0	18
	Sum	17	13	30	8	5	73
Code related	A	4	5	0	0	0	9
	B	6	12	14	0	0	32
	C	4	4	1	0	0	9
	D	3	3	8	0	0	14
	Sum	17	24	23	0	0	64
	Total	34	37	53	8	5	137
	(%)	(25%)	(27%)	(39%)	(6%)	(3%)	(100%)

Table 8
Non-code related problems categorized by origin.

Sys.	Arch.	Defects	Dev.	Ext. services	Infra-struct.	Runtime Env.	Spec	Sum
A	0	0	4	1	9	5	0	19
B	0	0	2	0	1	4	0	7
C	0	4	3	1	17	3	1	29
D	6	0	3	1	5	2	1	18
Total	6	4	12	3	32	14	2	73

4.2. Problems not directly associated to source code

The problems that were not directly associated with source code characteristics were attributed to a heterogeneous collection of origins that included: lack of adequate technical infrastructure (e.g., lack of proper data persistence frameworks), developer's strategy to solve a task, availability of external services (e.g., the web services provided by Simula's CMS), dependence on the runtime environment (e.g., problems due to conflicting versions of the Java runtime environment or JRE), and defects initially present in the system. Table 8 presents the distribution of the non-code related problems over the various systems, and Table 9 provides examples of the subcategories of this problem type.

It is worth noting that more than half of the problems did not relate directly to source code, with Inadequate infrastructure being the most salient one. This was particularly visible in Systems A and C (see Table 8). The original developers of these systems did not make use of adequate libraries for implementing data storage and search technology in their solutions. For example, the systems contained extensive and complex SQL queries 'hard-coded' within the Java code for performing different searches. The maintainers in our study found it extremely difficult to understand and modify these SQL queries, in particular for accomplishing Task 1. We note that 'hard-coded logic' could constitute an code smell (e.g., Excessive use of literals), but does not comprise any of the definitions of code smells we used for the purposes of this analysis, and consequently, problems due to this aspect were considered as non-code related problem.

In addition, System C had no clear separation of concerns between the presentation and business layers. This forced the developers to work with the business logic located in the JSP files, and perform modifications in a "manual" way, as they were deprived from much of the functionality in Eclipse that was only available for Java files.

Some of the problems were caused by the maintainer's approach to the solve the tasks and his/her coding style. Our observations showed that some of them had a high tendency to copy-paste code and not be very systematic in making the required adaptations. There were many occasions in which faults were introduced due to mistakes during these operations.

Another large portion of the problems were due to incompatibilities with the Java Runtime Environment (JRE), since the case study systems were originally developed for a much older version of the JRE (in 2004). The usage of deprecated components in some of the systems contributed to the problems, in particular during the *configuration stage* (i.e., installation and configuration of the working environment for the maintenance, prior the project kick-off).

Architectural issues manifested in System D, where developers were slightly taken aback by an additional layer within the

Table 9
Examples of non-code related problems.

Origin	Examples of problems faced by developers
Architectural challenges	Wasted a lot of time on useless refactoring involving 15 classes. Confusion about 4 tier system which has a "handler" and "command" tier. Working with an extra layer in the business logic produced delays.
Initial defects in the system	"Too many defects in the system, but only I noted them later on. This made estimations highly optimistic" (Bug in redirect pages caused delays during testing. Developer had to correct those before continuing.)
Developer factor	Lack of knowledge about technology forced re-development from scratch. Negative side-effects from developers' own copy-paste "strategy".
Availability of external services	Low performance of web services required for the maintenance tasks, made the testing and development slow.
Inadequate nfrastructure	Confusions due to several tables with duplicated content in the DB. Large SQL queries embedded in source were difficult to understand and debug. Copy-paste errors/inconsistencies in SQL queries induced defects and delays. Hard-coded class paths and redirects in the web pages were causing defects. Erroneous exception handling in JSP files made debugging difficult. Excessive logic embedded in JSP files made changes and debugging difficult. Excessive changes required on style-sheets demanded time.
Runtime environment challenges	Java Runtime Environment was incompatible with web service library (xmlrpc). Library compilation issues induced delays. IDE had limitations for handling different types of files. Delays due to problems configuring error logging on the server. (log4j) Delays due to problems configuring web servers (Tomcat).
Unclear specification	Misunderstanding of the maintenance requirements specification.

business logic layer (i.e., distinguishing between a "handler" layer and a "command" layer to enable that the web based presentation layer could be replaced by a standalone library in the future). As a result of this 4-tier design, there is a set of handler packages, which mirrored the command packages containing a set of functional entities. The developers complained about the fact that it was required to add two classes for each new functional unit to be added to the system, because of these two tiers. Although this was not considered very problematic, one of the developers tried to 'eliminate' this extra-layer, but had to rollback, incurring delays for the project.

It is worth noting that most of these problems could not have been detected via automated source code analysis (including code smell detection), which provides some insights on the scope and coverage level of code smells for assessing system-level maintainability. In addition, given that these four systems constitute web-applications, it is natural that many of the problems developers experience are not directly related to source code. However, many factors are in common with systems that are not web applications, such as Runtime Environment, External services and Developers.

4.3. Problems associated to source code

Problems associated to source code were grouped into three groups: 37 (58%) problems were attributed to the presence of code smells, 19 (30%) problems were attributed to other code characteristics, and 8 (12%) problems were the result of a combination of code characteristics, some of them including smells.

Table 10 shows a classification of all code-related problems, distinguishing those that were (solely) attributed to code smells, those that were attributed to other characteristics, and those that were attributed to a combination of factors, including code smells. The columns in the table classify the source code-related problems according to the type of problem (i.e., introduction of defects, program comprehension and time-consuming changes). First, we will describe those problems caused by code characteristics other than code smells. Second, we will describe the problems related to the presence of code smells. Finally, we describe those which were caused by a combination of factors, including code smells.

4.3.1. Problems caused by other characteristics than code smells

Many code-related problems could not be explained by the presence of code smells. Our analysis showed that this held for 30% of the source code-related problems. This indicates that alternative source code analysis techniques are needed in order to determine potential problematic areas in the code that need improvements. For some of the problems identified, alternative techniques such as semantic analysis [80] and dependency analysis [81] are available.

Table 10
Maintenance problems related to source code.

	System	Defects	Program comprehension	Costly changes	Total
Other code charact.	A	1	3	0	4
	B	0	6	4	10
	C	0	1	0	1
	D	0	0	4	4
	Sum	1	10	8	19 (30%)
Code smells	A	3	2	0	5
	B	5	6	3	14
	C	4	3	1	8
	D	3	3	4	10
	Sum	15	14	8	37 (58%)
Comb. factors	A	0	0	0	0
	B	1	0	7	8
	C	0	0	0	0
	D	0	0	0	0
	Sum	1	0	7	8 (12%)

4.3.1.1. (Lack of) Conceptual integrity. Inconsistent or illogical naming was considered problematic, in particular for System A. Developer 1 did not understanding why a file was called "StudySortBean" when its main responsibility was to connect a study to a researcher. These inconsistencies cannot be detected by code smells alone, and need manual inspection or semantic analysis. Lack of conceptual integrity was observed to introduce defects in the code and hinder program comprehension.

4.3.1.2. Internally complex code. Developers working with System B complained about side effects from modifying code which did not contain any code smells at all. One such class was examined and it was found that it implemented an extremely high number of methods (78 in total) and displayed intensive coupling (22 incoming dependencies) with another problematic class which

Table 11
Difficulties associated the introduction of defects.

Reason	# Obs.	Explanation	Code smell
Efferent coupling	2	Classes contained many methods that accessed data/methods from different areas of the system. Faults occurred because developers missed areas on the code that needed to be consistently changed after changes were done on the classes displaying the efferent coupling. This is a very similar situation to when consistent changes need to be done across duplicated code to avoid the introduction of defects	Feature Envy
Afferent coupling	3	When developers introduced faults in files with wide afferent coupling, the consequences of these faults will manifest across different components that are depending on them. This situation made many functional areas in the system to stop working after changes, and also led to unmanageable error propagation. The classes and interfaces with afferent coupling were recognized by the presence of ISP Violation and to some extent by Shotgun Surgery	ISP Violation
Large and complex classes	7	One or two classes in each system "hoarded" the business logic/functionality of the system. They were extremely large in comparison to the rest of the files of the systems. Developers frequently will commit "slips" or mistakes because of the size of these classes (mainly because it is hard to navigate across the class and keep track of changes within the class). These files will sometimes contain other smells such as Feature Envy, ISP Violation, and Temporal variable used for several purposes, but they will tend to be God Class or have multiple instances of God Method	God Class and/or God Method
Inconsistent variables	3	Some of these "hoarders" also contained Temporal variable used in different contexts which propitiated mistakes that led to several faults, particularly in System C. Developers would change a variable expecting that it will be used in the same way across the class, but instead unexpected behavior would manifest and demand debugging and refactoring	Temporal variable used in different contexts

constituted a God Class. The detection strategy for God Class used by Borland and InCode did not recognize the first mentioned class as a God Class. We conjecture that this occurred because the detection strategy considers the size of the methods (i.e., not only the total size of a class), and in this particular class, all 78 methods were very small.

4.3.1.3. Files with cyclic dependencies. In System B, many of the classes that exhibited program comprehension issues contained cyclic dependencies that hindered the understanding of the overall system's functionality.

4.3.1.4. Multiple inheritance and dynamic binding. In System B, the implementation of multiple interfaces to simulate multiple inheritance (see [82] for detailed explanation) led to such complex (and dynamic) dependencies between classes that it prompted one of the developers to remove code that he erroneously considered "dead code". After finding out that it was not, considerable effort was needed to roll back this change. Currently, there is no code smell definition related to "Multiple inheritance simulation" but we propose it as a new smell, given its potentially serious consequences.

4.3.1.5. External libraries. In System B, maintainers spent many hours trying to understand how to use the complex proprietary persistence framework that was used during initial development. This was considered one of the main reasons why changes were costly or expensive in System B. They also had serious issues with the Tomcat Realm library, which is an authentication mechanism that manages usernames and passwords for valid users of a web application and controls the roles associated with each valid user.

4.3.1.6. Implementation shortcomings. System A had a problem relating to the Java language version used at the time of initial development. The system did not use Java Generics[11] (see [83] for more information) when defining Map and Hashtable types. During maintenance, this use of non-typed Maps and Hashtables required manual identification of all areas of the code where the Employee or Publication IDs had to be changed from Integer to String. Since the maintainers sometimes missed locations where types should have been replaced, this resulted in defects after Task 1. A related problem was found in System D: the entire system used a Hashtable ⟨*Integer*,*String*⟩ to transfer data between the business layer and the presentation layer. As a result, developers could only send an Integer (ID) and a String (which in this case was the title of the publication) to the presentation layer. Although maintainers wanted to transfer more information on Publications, this decision was so entangled in the system that they were forced to use workarounds instead of simply modifying the Hashtable to support a more generic type.

4.3.2. Difficulties associated to code smells

Several problems that occurred during maintenance could be explained by occurrences of code smells in the problematic code. For each type of problem described earlier in Table 6 (i.e., defects introduced, difficult program comprehension and costly changes), we will now discuss how they occurred, and which code smells were detected in the problematic code that can explain those issues.

4.3.2.1. Introduction of defects. We found four main code characteristics that according to our observations, led to the introduction of defects after changes: (1) wide efferent coupling, (2) wide afferent coupling[12], (3) large and complex classes, and (4) inconsistent use of variables. Large and complex classes were the main reason for most of the problems registered under this group (7 observations in total). Files belonging to each of these four groups were characterized primarily by the presence of several code smells: Feature Envy, ISP Violation, God Method/God Class and Temporal variable used for several purposes. Table 11 shows for each code characteristic, the number of observations, an explanation why the problem occurred, and the most salient code smell across the files associated to those problems.

4.3.2.2. Troublesome program comprehension and information searching. Table 12 presents the various reasons that we observed to underlie maintenance problems related to program comprehension and information searching. As mentioned previously, there were different levels of program comprehension problems

[11] Java Generics is a facility for generic programming introduced to the Java as part of J2SE 5.0. Generic programming is a style of programming in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters.

[12] *Efferent coupling* is a measure of how many different classes are used by the specific class, also known as outgoing dependencies, and *Afferent coupling* is a measure of how many other classes use the specific class, also known as incoming dependencies.

Table 12
Reasons and smells underlying observed maintenance difficulties that were related to program comprehension and information search.

Reason	# Obs.	Explanation	Code smell
Cross-cutting concerns	2	In both Systems A and B, the domain entity Person was being used within the "User" context and also within "Employee" concept. As such, information and functionality connected to the studies (e.g., employees who participated on it) and the management of privileges and access constituted crosscutting concerns. In System A, the entity person had many clients, thus resulting in ISP Violation and in System B, the entity person was hoarding all the functionality that related to both concepts employee and user	Feature Envy or ISP Violation
Inconsistent Design	1	Imbalances and inconsistencies in the allocation of data and functionality across classes confused a developer. 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. In System A, a class was acting as both Bean and Action file, and it was accessed by a wide set of dissimilar clients. This confused one developer, who was not sure about the intended scope of the code, which led to delays in the project, because the developer spent time deciding on the right strategy to follow in order to solve the task	Data Class and ISP Violation
Inconsistent variables and duplication	3	Inconsistencies in the use of many different variables made the developers unsure about the purpose of the variables, and the behavior of the method/class containing them. Also duplicated code or similar functionality implemented inconsistently made it difficult to understand the code	Temporal variable used in different contexts
Large and complex classes	2	Files that were hoarding the functionality of the system were extremely big in size, and consequently, difficult to navigate within. This resulted in developers struggling to understand the roles and behavior of these classes. Some of them would display wide afferent and efferent coupling, but the biggest distinguishing factor was the presence of God Class and/or God Methods	God Class and/or God Method
Pervasive classes	3	Classes which acted as a "Middle Man" in every functional aspect of the system made it difficult for the developers to grasp the overall behavior of the system. System B had a Memory caching mechanism implemented through one class which was involved in every data-related operation in the system. Developers spent hours investigating this mechanism before feeling enough confident to perform any changes	God Class
Logic Spread	2	Classes that contained many methods making many calls to methods or variables in other classes, forced the developers to examine all the files called by these methods in order to identify the pertinent data or location where to perform the change. Some of the God Methods also called other God Methods, which made the problem worse	God Method and Feature Envy

identified during the project. The presence of one pervasive class (named MemoryCachingSimula) caused most of the problems in understanding the overall behavior of system. This class constituted a God Class Confusion during program comprehension was caused by inconsistent design and cross-cutting concerns, which can be explained by the presence of Data Class and ISP Violation. Files that developers associated with inconsistent use of variables displayed the smell Temporal variable used in different contexts. We could observe two cases of cross-cutting concerns, which were explained by afferent (ISP Violation) and efferent coupling (Feature Envy). Finally, problems understanding low-level details of the code were caused by large classes and wide spread logic, explained by a combination of God Method and Feature Envy.

4.3.2.3. Costly changes. The observations in this category are related to highly complex changes, both in terms of number of changes required to complete the task, as well as in terms of the number of elements that a developer needs to consider simultaneously in order to complete the task (i.e. a cognitively demanding task). The code smell that was most salient within this group was Feature Envy, followed by ISP Violation and God Class (See Table 13). Most of the files associated with costly changes displayed a combination of God Class and Feature Envy. We observed that many files associated with complex changes were also associated with the introduction of defects. It is plausible that these complex tasks have led to the introduction of faults, since the developers could not keep track of the potential consequences of their changes due to the complex design.

4.3.3. Difficulties associated to interaction effects

We observed that a single characteristic could lead to different types of problems, but we also found instances of interaction effects in this study. In particular, we identified a set of files (marked with bold) that exhibited a combination of different characteristics, and they were associated with multiple problems during maintenance. These files (classes) also showed a higher number of churn and number of revisions than the average for a file in a given system (see Table 14). In Table 14, we mainly see two sets of files: One set that contains either zero code smells or very few (ex. ObjectStatement, PeopleDAO), and another set comprising extremely large files displaying a combination of smells: Feature Envy, God Method, ISP Violation, Shotgun Surgery and Temporary variable used for several purposes (ex. StudyDatabase, DB, StudyDAO).

We could observe that the files with none or nearly none smells were coupled to files containing many code smells, and we identified several cases where particular characteristics in the former group of files interacted with the smells present in the latter group of files, causing problems to developers.

We named the files containing a lot of code smells "control hoarders", mainly because they contained the majority of the functionality. Each system except for System B displayed one instance of them (i.e. DB in System C, StudyDatabase in System A, and StudyDAO in System D), and they were associated to all types of problems, and were unanimously mentioned by all developers that worked with the same system. They all contained: Feature Envy, God method, and ISP violation, and, in addition they also exhibited high churn rate and LOC. We found that in System B, the previously mentioned combination of smells was not located in one file, but distributed across several problematic files. StudySearch and MemoryCachingSimula were internally complex, whiles Simula and ObjectStatementImpl displayed a wide spread afferent/efferent coupling. Both pairs of files were coupled (i.e., MemoryCachingSimula had dependencies on Simula while StudySearch had dependencies on ObjectStatementImpl).

We could observe in our study that two coupled files can contain the same combination of smells (distributed over the files) as one single file, and lead to the exact same consequences. This means that for a practical perspective, there is no difference if a combination of smells is located in one file or if it is distributed across coupled files.

Table 13
Difficulties associated with costly changes.

Reason	# Obs.	Explanation	Code smell
Lack of flexibility of dependent components	3	Classes that contained many calls to different elements were dependent of the implementation on those elements. When developers introduced changes in a class, the elements on which the class was relying on will not allow the changes due to their lack of flexibility in their design. This forced the developers to roll-back changes and try out different solutions/strategies to complete a task	Feature Envy
Efferent coupling	4	When classes call many methods (in particular for data access) from other classes (delocalized logic), this led to "cognitively demanding" changes. This became particularly problematic when trying to merge the data retrieved from the new web services and the data retrieved from the DB	Feature Envy
Afferent coupling	1	When changes were introduced to interfaces with wide dependencies, adaptation or amends were needed on those elements depending on the interfaces. This led to time-consuming change propagation	ISP Violation
Large and complex classes	1	When classes are very big and contain many methods without clear separation of concerns, parallel changes were required in each of them	God Class + God Method

Table 14
Files linked to problems due to code smells or to a combination of characteristics.

System	File	obs	DC	CL	DUP	FE	GC	GM	ISPV	MC	RB	SS	Temp	Imp	rev	chrn	loc
A	PublicationDatabase	2	0	0	0	2	0	0	0	0	0	0	1	0	8	557	308
A	StudyDatabase	2	0	0	0	7	0	1	1	0	0	1	1	1	23	525	888
B	**ObjectStatement**	2	0	0	0	0	0	0	0	0	0	0	0	0	2	4	133
B	PrivilegesManageAction	2	0	0	0	0	0	1	0	0	0	0	1	0	9	304	225
B	Publication	2	1	0	0	0	0	0	0	0	0	0	0	0	8	190	84
B	Simula	2	0	0	0	0	0	0	1	0	0	1	0	0	12	806	408
B	ConfigServlet	3	0	0	0	0	0	0	0	0	0	0	0	0	5	176	152
B	MemoryCachingSimula	3	0	0	0	0	1	0	0	0	0	0	1	0	13	1011	611
B	ObjectStatementImpl	3	0	0	0	0	0	0	1	0	0	1	0	0	9	1134	796
B	Person	3	0	0	0	0	0	0	1	0	0	1	0	0	14	451	193
B	StudiesEditAction	3	0	0	0	0	1	2	0	0	0	0	0	0	10	484	368
B	**Persistable**	4	0	0	0	0	0	0	0	0	0	0	0	0	2	2	22
B	**PersistentObject**	0	0	0	0	0	0	0	0	0	0	0	0	0	3	2	70
B	StudySearch	4	0	0	0	2	1	2	0	0	0	0	0	0	10	1213	786
C	DB	9	0	0	2	16	1	2	1	0	0	0	1	0	34	2754	1824
D	PublicationDAO	2	0	0	0	0	0	0	0	0	0	0	0	0	12	336	159
D	PeopleDAO	3	0	0	0	0	0	0	0	0	0	0	0	0	16	372	261
D	StudyDAO	8	0	0	1	10	1	2	1	0	0	1	1	0	30	480	1268

The following example was identified in System B, where a particular combination of factors led to costly changes and defects were introduced after changes. This issue was reported by all developers who worked in System B and it manifested through 8 different maintenance problems.

4.3.3.1. Critical example of an interaction effect. The developers who worked on System B wanted to replace two interfaces: Persistable and PersistentObject with one new interface that would support a String ID type instead of an Integer type in order to complete Task 1. This was not possible since the entire implementation was based on primitive types instead of domain entities. Both interfaces were restrictive and were made under the assumption that identifiers for objects would always be Integers, and thus defined accessor methods getId () and setId () with Integer types. Note that these interfaces displayed *zero* code smells. The problem occurred because many critical classes in the system implemented these two interfaces, consequently the change spread was extensive. See for example in Fig. 3, how up to 10 domain classes implemented the PersistentObject interface. Many of the classes that implemented these interfaces displayed ISP Violation, which resulted in an extensive ripple effect. Fig. 4 shows how ObjectStatementImpl (which displayed *both* Shotgun Surgery and ISP Violation) displayed dependencies on the Persistable interface. Due to these dependencies, from a practical perspective modifying Persistable would have the same impact (in terms of ripple effect) as making modifications directly in ObjectStatementImpl. Modifying any of these interfaces led to an unmanageable number of compilation errors, so developers had to rollback the changes in those files (i.e. keep the interfaces untouched) and instead perform forced casting wherever it 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 code smells intensify or spread the effects of certain design choices throughout the system. The above-mentioned interfaces *alone* where not so harmful, in spite of having a design flaw (i.e., having typed setters and getters). But the moment that classes widely used in the system display a dependency on them, that would spread the effects of this design limitation.

5. Discussion

5.1. Code characteristics (and smells) that can lead to maintenance difficulties

Our results indicate that there are many reasons why difficulties can occur during maintenance, and code smells can help to explain and potentially identify some of those problems beforehand. However, as we observed several cases where distinct problems were caused by the same code characteristics, the predictive capacity will generally be limited to signalling that problems *are* to be expected, but not *what* concrete problems these are.

In our study, we identified four code characteristics that are strongly related to current code smell detection strategies, that explained maintenance difficulties: (a) Inconsistent design

Fig. 3. Diagram of interface dependencies of PersistentObject.

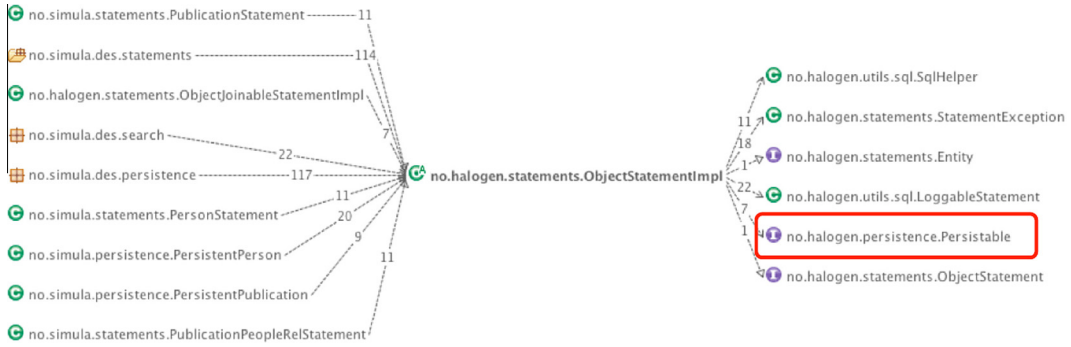

Fig. 4. Incoming/outgoing dependencies of ObjectStatementImpl.

(incl. usage of variables), (b) cross-cutting concerns, (c) large and complex classes, and (d) coupling (afferent and efferent).

5.1.1. Inconsistent design

This characteristic was found to negatively affect program comprehension and in some situations led to the introduction of faults. Although design inconsistencies can manifest in a semantic form (as discussed in Section 4.3.1), they also manifest through imbalances in the data/functionality allocation. Our observations on System A suggest that this characteristic can be identified by detecting a combination of Data Class and an ISP Violation smells. We also found design inconsistencies that manifested at the variable level, related to Reuse of Temporary Variable for unrelated purposes.

5.1.2. Cross-cutting concerns

This characteristic was found in this study to negatively affect program comprehension (See Section 4.3.2, paragraph: *Troublesome program comprehension and information searching*). As concerns can pass through different layers or packages, they can manifest in the form of widespread dependencies. As such, either Feature Envy (which indicates efferent coupling) or ISP Violation (which indicates afferent coupling) can help to identify these situations, in combination to approaches for identifying cross-cutting concerns.

5.1.3. Large and complex classes

This characteristic was attributed as the cause for all three major types of difficulties (i.e., introduction of defects, troublesome program comprehension and costly changes), and it largely relates to the size factor. We observed that most of the files with large size exhibited either the God Class smell or the God Method smell and in most cases a combination of both smells. This finding is confirmed by the results from Abbes et al. [42]. When it comes to program comprehension difficulties, our results are in agreement to observations by Deligiannis [39], who found that designs that exhibit God class led to reduced correctness and increased effort in maintenance tasks. In our study, many of the large and complex classes also contained other smells such as Feature Envy, ISP Violation and Shotgun Surgery (see Table 14). This suggests that these difficulties are caused by a indirect consequence of size (i.e., they can be caused by interaction effects between code smells that are more likely occur in large classes). For example, Large classes (or God Classes) often are difficult to navigate through and read. But if they contain in addition, many Feature Envy methods, that would cause additional difficulties, because it will make time-consuming or even confusing, the navigation *across* artifacts displaying dependencies (e.g., methods or variables in other classes that are accessed by the Feature Envious methods). What happens is that Feature Envy methods tend to appear in classes that contain high amounts of logic. We observed such cases in our study, where we found that is not the actual *size* of the artifact, but also the large number of outgoing dependencies that *co-occur* in these large artifacts, that intensified the problems. Olbrich et al. [45] reported that when normalized with respect to size, classes constituting God Class or Brain Class had less faults and demanded less effort. The results on our study suggest that observing combinations of code smells could be useful to discriminate instances of God Classes that

are potentially more problematic (by for example focusing on those God Classes which also display Feature Envy, ISP violation and Shotgun Surgery).

5.1.4. Coupling

Afferent and efferent coupling were associated to all three major types of difficulties (i.e., introduction of defects, troublesome program comprehension and costly changes) in our study. We found that classes with wide coupling displayed Feature Envy or ISP Violation and to some extent Shotgun Surgery. Li and Shatnawi [35] reported that Shotgun Surgery was positively associated to faults, and in our study we observed several situations where afferent coupling led to the introduction of defects.

Also, the nature of the maintenance task created a situation, where "consistent changes" were needed (*e.g.,* all the functionality related to Employee and Publication should use String IDs instead of Int IDs). This situation is very similar to what happens when duplicated code needs to be consistently and simultaneously updated: when it's not, it leads to defects (see work by Juergens et al. [36]). When the logic is widespread over the code, it will become easier for developer to overlook places where changes/updates need to be made, leading to defects. To address this issue, either the design of the system or appropriate tools should support efficient localization of the task context and change propagation.

From a program comprehension perspective, our observations challenge the current views on how delocalized plans only affect unexperienced developers (as reported by Arisholm and Sjøberg in [84]). Although the developers in our study were as experienced as in [84], in our study, they were observed for a longer period of time, and they worked with more complex tasks and systems than in the study reported in [84]. An explanation on why coupling turned out to be problematic for program comprehension in our study, and not in the study reported in [84], can be due to the presence of thresholds on the levels of delocalized logic that developers can handle without the task becoming too cognitively demanding.

An insightful case was System B, which in order to successfully complete Task 3, required the combined activities of understanding and (re)using certain libraries that were build by the original developers of the system (i.e., a library for generating requests to the DB, which was needed to produce the reports). What developers referred as "time-consuming" or costly work was to actually put the elements provided by the library together in order to solve the task. Our observations are aligned with Wood's [85] perspective on component task complexity, referring to the number of distinct information cues that must be processed simultaneously to perform a task. We could identify cases where this component complexity increased the effort on code comprehension and modification, as developers needed to gather the distinct information cues to complete the task.

5.2. Challenges of code smell analysis due to interaction effects

Within the categorization made on the maintenance difficulties, there is a degree of overlap between the difficulty category described in Section 4.3.2 (i.e., problems attributed to the presence of code smells) and the category described in Section 4.3.3 (i.e., problems due to interaction effects). However, we believe the distinction necessary in order to build more detailed causal models for attempting to understand the relationship between code attributes and maintenance.

For example, in Section 4.3.3, it was described how modifying two widely used interfaces in System B led to unmanageable error propagation. The case portraits what happens when a widely used interface (manifested in the form of ISP violation) that has a limited design choice (i.e., the definition of getters and setters using primitive types) needs to be modified. We could observe that there was an interaction effect between these two characteristics, which in the context of the maintenance task (i.e., Task 1: replacing ID type of the domain entities from Integer to String) resulted in a considerable problem for the developers. We believe this is a very important observation to report, as it could be the key to actually understand why code smells are harmful in certain situations, and in some other, they are not. This may explain why different empirical studies report contradictory effects on the same code smell.

In our study, we observed interaction effects between a code smell and other code characteristics, but we also observed interaction effects between code smells. Pietrzak [35] introduced the notion of inter-smell relations. An example of an inter-smell relation is plain-support: "... smell B is supported by smell A if the existence of A implies with sufficiently high certainty the existence of B. B is then a companion smell of A, and the program entities (classes, methods, expressions, etc.) burdened with A also suffer from B..." (p. 77) [86]. Pietrzak suggest the notion of inter-smell relations to support more accurate code smell detections, but it can also be used to better understand the potential interaction effects due to different combinations of code smells.

We have identified and reported explicit cases where the presence of a single code smell in a file did not lead to as much problems as when several specific code smells, where present, suggesting that interaction effects between code smells exist. If we have grouped the code smell interaction cases (reported in Section 4.3.3) together with the first category (described in Section 4.3.2), we would have omitted an important cue, as we would assume that code smells that appear alone have the same effects as those that interact with other smells or other code characteristics. Very little has been investigated in the current literature on the interplay between code smells, being the work by [42], the only one reporting on potential interaction effects (i.e., between God Method and God Class). Thus, we believe that correlation studies involving individual variables are perhaps, not enough for understanding the effects of code smells in a comprehensive manner.

In Section 4.3.3 we discussed how in System B, interaction effects occurred across code smells not *collocated* in a single file, but distributed across several files that were *coupled*. Up to now, code smells have been detected mainly at class or method level. Considering the fact that the consequences of such *coupled* smells are the same for *collocated* smells, we can assume that there is a need to consider smell interactions across artifacts that are coupled. To include such interactions, code smell detection algorithms should not be limited to class or method levels, but also consider the dependencies between artifacts.

5.3. Capability of code smells to predict system's overall maintainability

This paper has provided a detailed account on the different aspects of the four systems that caused maintenance difficulties. This gives us an insight in the plethora of factors that may influence maintenance. Several of these factors have been reported before [59,57,87], but here we provide a detailed, systematic analysis based on an industrial case study that included observing developers conduct maintenance tasks over a longer time.

From the total set of difficulties identified during maintenance, less than half (43%) were related to Java code, and from those, only 58% clearly related to any of the twelve code smells used to analyze the code. This means that even if we count those difficulties that are due to combination of factors, roughly only 30% of the total set of difficulties can be explained and potentially foreseen by code smells. As a result, we conclude that the subset of aspects that *are* covered by current code smell detection has a relatively low

impact on the outcomes of a maintenance project, and code smell detection is of limited value as single predictor of maintainability.

Moreover, our results show that there are several code characteristics that are not captured by code smells but did affected a considerable part of maintenance difficulties (30%). This suggests that a combination of different code analysis techniques and tools is needed in order to achieve a more comprehensive evaluations of maintainability. This aligns with the ideas of Walter and Pietrzak [88], who suggested the usage of multiple criteria to assess code quality.

There were some differences in the types and number of problems across systems, for example System B displayed more code-related problems (82% of the total), whiles System C displayed more non-code related problems (76% of total problems). One can then argue that the proportion of java files versus other types of files (ex., jsp, sql, php) may influence the proportion of code vs. non-code related maintenance problems (89% of the files in System B were java files, whiles in System C, java files constituted only 46% of the total system).

However, we observed that proportion of files does not always explain the type of problem: whiles in Systems B and C, the proportion of code-related problems is in accordance to the proportion of Java files, Systems A and D, displayed more non-code related problems (32% and 44% respectively) despite the fact that they contained a considerably higher number of Java files than Jsp or other files (74% and 79% respectively). This suggests that the sources of the maintenance problems identified in our study constitute a broader range of aspects than the proportion of files in the business and presentation layers.

The results from our study reminds us of the limitations of evaluations based purely on static analysis and suggest the need of more comprehensive quality models and techniques that can incorporate the analysis of diverse factors. This position was also taken by Anda [87], who suggests that maintainability evaluations should combine the analysis of code measures with expert assessment, as they both cover different aspects of software quality.

5.4. Threats to validity

We consider four perspectives in the threats to the validity of our study:

5.4.1. Construct validity

Maintenance problem constituted a simple but straightforward construct in our study. Although this term has not been previously defined in the literature, we provide a comprehensive description of the problems. With respect to code smells, we used automated detection to avoid subjective bias. Nevertheless, the lack of standard detection strategies used in the tools could be a potential threat, as they can affect some of the results reported in this study. For example, we found classes with complex code were not identified God Class, and did not contain Feature Envy, or Long Method smells. 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.

5.4.2. Internal validity

In this study, we have used mainly qualitative analysis techniques to explore potential causal links between code smells and maintenance problems, and their corresponding underlying causal mechanisms. According to Maxwell [89], there are mainly two ways to derive/test theories and investigate causality: *variance theory*, which involves measurements of differences and correlations. In contrast, there is an approach called *process theory*, which deals with events and processes that connect them, is less compatible to statistical approaches, and focuses instead to in-depth study of a limited number of cases [89]. Maxwell states:

> "Both types of theories involve causal explanation. Process theory is not merely *descriptive*, as opposed to *explanatory* variance theory; it is a different approach to explanation. Experimental and survey methods typically involve a *black box* approach to the problem of causality; lacking direct information about social and cognitive processes, they must attempt to correlate differences in output with differences in input and control for other plausible factors that might affect the output. Qualitative methods, on the other hand, can often directly investigate these causal processes, although their conclusions are subject to validity threats of their own".

In this study, we have used a technique called "explanation building" where we provide the sequence of events, factors or situations that lead to a given problem. As in any qualitative research, there are threats to validity and limitations. One limitation is that the level of detail on the narrative is not homogeneous for all the problems mentioned. A threat is that is possible that some developers were more open about problems than others, and some did not report all the maintenance problems they experienced. As with most other qualitative research, researcher bias may occur when selecting data to analyze and report and when summarizing and interpreting the data. Maxwell and also Yin [56], suggests a series of approaches or characteristics necessary in a qualitative inquiry to tackle its threats to validity. Our study possess many of important characteristics reported, in specific:

- *Intensive, long-term involvement*, which according to Becker and Geer [90], as cited by Maxwell "provides more complete data about specific situations and events than any other method".
- *Rich data*, which according to Maxwell, given the condition that is detailed and varied enough, they can, provide a full and revealing picture of the processes involved.
- *Searching for discrepant evidence and negative cases*. The use of the *modus operandi* across cases was applied in our study to strengthen the causal inference on the effect of code smells. For example, the fact that all developers who worked with the same system faced the exact same problem, strengths the validity of the narrative describing the process involved in the causal relations.
- *Member checks*. During the study, feedback was requested from other researchers on the data being collected, and in the latter stage where the analysis and the writing of this paper was conducted, there was a intensive discussion on the nature of the problems, the evidence and the categorization.
- *Triangulation*. "Collecting information from a diverse range of individuals and settings or using a variety of methods" [89], via the usage of three independent collection methods, i.e., interviews, direct observation and think-aloud sessions.

5.4.3. External validity

The main threat to external validity of this study is that the results are contingent to the contextual characteristics and industrial domain of the project, encompassing a Java web-based, medium-sized (10–30KLOC), three-layered information system. Had the project not involved a web-application, the number of code smells detected could have been different, and the percentage of problems that could be connected to code smells could have been different. Also, it is possible that not all the definitions of code smells used are equally applicable in every domain. For example,

Table A.15
Difficulties, description and files associated.

ID	Description (Summary)	Code?	Factors	File	Dev.	Sys.	Diff. type
40	Slightly confused about 4 tier system which has a "handler" and "command" tier	no	Arch.	n/a	3	D	Program compreh.
49	Working with handlers and jsp took a lot of time	no	Arch.	n/a	3	D	Modifying
9	Query *study_search_query* is difficult to understand	no	Infrastr.	n/a	1	A	Understanding
11	Difficulties with usage of wildcard "%" in sql queries	no	Infrastr.	n/a	3	C	Understanding
37	Copy-paste error in sql query took around 2 h	no	Infrastr.	n/a	1	A	Defects
57	Complaints about the size of the query	no	Infrastr.	n/a	2	A	Understanding
21	Problems after adding the personalized report functionality	no	Infrastr.	n/a	3	C	Defects
25	Bug due to userId which was left without conversion from int to String	no	Developer	ReportListAction.java, ReportDefinitionsEditAction.java	2	B	Defects
2	Dynamic binding made programmer think erroneously that code was dead	yes	n/a	SearchByPublicationTitle.java, SearchByPublicationTitleImpl.java, CriteriumFactoryImpl.java, PublicationStatement.java	2	B	Program compreh.
3	Finding task context is difficult for Person and Publication. Specially person since domain is localized but code is spread	yes	n/a	Person.java	2	B	Program compreh.
4	Hard to find entity People	yes	n/a	PeopleDatabase.java	1	A	Program compreh.
8	Programmer does not understand if User is creator of Publication	yes	n/a	PublicationDatabase.java	1	A	Understanding
10	Difficulties finding the assignment of the results due to polymorphism in the statement	yes	n/a	PersistentPerson.java	2	B	Understanding
12	Variable "Search" used in different contexts, make it difficult to understand	yes	n/a	DB.java	3	C	Understanding
15	Bug due to temporal variables repeatedely used for different purposes	yes	n/a	DB.java	3	C	Defects
20	Difficulties understanding the caching system	yes	n/a	MemoryCachinSimula.java	2	B	Understanding
22	Hard to understand the realm concept	yes	n/a	SimulaRealm.java	2	B	Understanding
23	Problems with understanding the previous authentication procedure	yes	n/a	SimulaRealm.java	2	B	Understanding
27	Casting problems (due to DB drivers), and this forced the programmer to do hard casting from long to int	yes	n/a	ObjectStatementImpl.java	2	B	Modifying
28	Defects introduced after modifying function create new study (Last modified/created by info lost during int-string conversion)	yes	n/a	StudiesEditAction.java	2	B	Defects

Table A.16
Code characteristics and files that were associated to the introduction of defects.

Reason	ID	Sys	File	DC	CL	Dup	FE	GC	GM	Ispv	MC	RB	SS	Tmp	Imp	rev	chrn	loc
Efferent coupling	58	A	RemoveResponsibleAction	0	0	0	1	0	0	0	0	0	0	0	0	5	23	80
	103	D	SearchCriteriaTag	0	0	0	1	0	0	0	0	0	0	0	0	4	41	51
Afferent coupling	97	D	Nuller	0	0	0	0	0	0	1	0	0	1	0	0	2	22	121
	98	D	StudyDAO	0	0	1	10	1	2	1	0	0	1	1	0	30	480	1268
	113	C	DB	0	0	2	16	1	2	1	0	0	0	1	0	34	2754	1824
	113	C	ReportServlet	0	0	0	0	1	0	0	0	0	0	3	0	10	184	441
Inconsistent variables	15	C	DB	0	0	2	16	1	2	1	0	0	0	1	0	34	2754	1824
	78	C	DB	0	0	2	16	1	2	1	0	0	0	1	0	34	2754	1824
	137	A	StudyDatabase	0	0	0	7	0	1	1	0	0	1	1	1	23	525	888
	137	A	PerformSearchStudiesAction	0	0	0	1	0	0	0	0	1	0	0	0	4	23	95
Large and complex classes	28	B	MemoryCachingSimula	0	0	0	0	1	0	0	0	0	0	1	0	2	1	13
	28	B	PersistentStudy	0	0	0	4	1	0	0	0	0	0	0	0	5	1	13
	28	B	PrivilegesManageAction	0	0	0	0	0	1	0	0	0	0	1	0	2	1	9
	28	B	StudiesEditAction	0	0	0	0	1	2	0	0	0	0	0	0	10	484	368
	32	B	StudiesEditAction	0	0	0	0	1	2	0	0	0	0	0	0	10	484	368
	39	B	StudiesSearchAction	0	0	0	1	0	1	0	0	0	0	1	0	10	40	262
	39	B	StudySearchForm	1	0	0	0	0	0	0	0	1	0	0	0	4	269	221
	70	C	DB	0	0	2	16	1	2	1	0	0	0	1	0	34	2754	1824
	70	C	StudyEditServlet	0	0	0	0	0	0	0	0	0	0	0	0	2	1	115
	77	A	StudyDatabase	0	0	0	7	0	1	1	0	0	1	1	1	23	525	888
	87	B	StudySearch	0	0	0	2	1	2	0	0	0	0	0	0	10	1213	786
	119	B	StudySearch	0	0	0	2	1	2	0	0	0	0	0	0	10	1213	786

Data Class constitutes a code smell according to Fowler [7], but it constitutes a well-accepted practice in web applications that use EJB (Enterprise Javabeans).[13] To achieve a better maintainability analysis based on code smells, we believe that future work should focus on defining code smells that match the domain of the systems under analysis, such as the work by Guo et al. [91].

In addition, the maintainers performed their tasks as solo-projects. The fact that these were solo-projects can affect the applicability of the results in highly collaborative environments. Despite these limitations, the tasks observed are based on real needs and their duration and complexity are representative of real-life projects. The tasks resemble backlog items in a single sprint or iteration within the context of Agile development. Studies of in vivo maintenance tasks have not reported more than 240 min, whereas

[13] http://www.oracle.com/technetwork/java/javaee/ejb/index.html.

Table A.17
Code characteristics and files that were associated to the difficult program comprehension.

Reason	ID	Sys	File	DC	CL	Dup	FE	GC	GM	Ispv	MC	RB	SS	Tmp	Imp	rev	chrn	loc
Cross-cutting concern	3	B	Person	0	0	0	0	0	0	1	0	0	1	0	0	14	451	193
	4	A	PeopleDatabase	0	0	0	3	0	0	0	0	0	0	0	0	12	359	282
Inconsistent functionality allocation	36	A	StudySortBean	1	1	0	0	0	0	1	0	0	0	0	0	6	60	154
Inconsistent variables and duplication	12	C	DB	0	0	2	16	1	2	1	0	0	0	1	0	34	2754	1824
	63	B	PrivilegesManageAction	0	0	0	0	0	1	0	0	0	0	1	0	9	304	225
	80	C	DB	0	0	2	16	1	2	1	0	0	0	1	0	34	2754	1824
	88	C	DB	0	0	2	16	1	2	1	0	0	0	1	0	34	2754	1824
Large and complex classes	42	D	StudyDAO	0	0	1	10	1	2	1	0	0	1	1	0	30	480	1268
	62	B	StudiesEditAction	0	0	0	0	1	2	0	0	0	0	0	0	10	484	368
Pervesive classes	20	B	ConfigServlet	0	0	0	0	0	0	0	0	0	0	0	0	5	176	152
	20	B	MemoryCachingSimula	0	0	0	0	1	0	0	0	0	0	1	0	13	1011	611
	20	B	PersistentFactory	0	0	0	0	0	0	0	0	0	0	0	0	3	8	87
	20	B	Person	0	0	0	0	0	0	1	0	0	1	0	0	14	451	193
	20	B	Privilege	1	0	0	0	0	0	0	0	0	1	0	0	4	100	72
	20	B	Publication	1	0	0	0	0	0	0	0	0	0	0	0	8	190	84
	20	B	Simula	0	0	0	0	0	0	1	0	0	1	0	0	12	806	408
	20	B	SimulaException	0	0	0	0	0	0	0	0	0	0	0	0	1	11	25
	20	B	SimulaFactory	0	0	0	0	0	0	0	0	0	1	0	0	0	0	40
	117	B	ConfigServlet	0	0	0	0	0	0	0	0	0	0	0	0	5	176	152
	117	B	MemoryCachingSimula	0	0	0	0	1	0	0	0	0	0	1	0	13	1011	611
	118	B	Table	0	0	0	0	0	0	1	0	0	1	0	0	1	1	190
Logic Spread	66	D	StudyDAO	0	0	1	10	1	2	1	0	0	1	1	0	30	480	1268
	75	D	StudyDAO	0	0	1	10	1	2	1	0	0	1	1	0	30	480	1268

Table A.18
Code characteristics and files that were associated to the difficult program comprehension.

Reason	ID	Sys	File	DC	CL	Dup	FE	GC	GM	Ispv	MC	RB	SS	Tmp	Imp	rev	chrn	loc
Lack of flexibility	43	D	StudyDAO	0	0	1	10	1	2	1	0	0	1	1	0	30	480	1268
	51	B	StudySearch	0	0	0	2	1	2	0	0	0	0	0	0	10	1213	786
	56	B	StudySearch	0	0	0	2	1	2	0	0	0	0	0	0	10	1213	786
Data dependencies	45	D	PeopleDAO	0	0	0	0	0	0	0	0	0	0	0	0	16	372	261
	45	D	PublicationDAO	0	0	0	0	0	0	0	0	0	0	0	0	12	336	159
	45	D	StudyDAO	0	0	1	10	1	2	1	0	0	1	1	0	30	480	1268
	46	D	PeopleDAO	0	0	0	0	0	0	0	0	0	0	0	0	16	372	261
	82	D	StudyDAO	0	0	1	10	1	2	1	0	0	1	1	0	30	480	1268
	108	B	ConfigServlet	0	0	0	0	0	0	0	0	0	0	0	0	5	176	152
Large classes	91	C	DB	0	0	2	16	1	2	1	0	0	0	1	0	34	2754	1824

in this study, we closely observed the whole maintenance process for a period of 120–160 h per system.

5.4.4. Reliability

Finally, with respect to the reliability (or repeatability) of our study: we have comprehensive documentation of the study protocol, the procedures for conducting the interviews, think aloud sessions and for summarizing and analyzing the data. Most importantly, the systems and the data used as baseline for this study bear enough complexity to belong to a realistic setting. Situations that normally do not occur in controlled settings were observed and provided new insights that can be useful for practitioners and researchers.

6. Conclusions

In total, 137 different difficulties were identified from the different maintenance projects, from which only 64 difficulties (47%) were associated to Java source code. The remaining 73 (53%) constituted difficulties not directly related to code, such as: lack of adequate technical infrastructure, developer's coding habits, dependence on external services (e.g., web services), incompatibilities in the runtime environment (e.g., JRE), and defects initially present in the system. Our results showed a relatively low coverage of code smells when we observe the maintenance project as a whole. Within the limits established by the context of this study, it was clear that the proportion of problems that were linked to source code and more specifically, to code smells was not as large as expected. A corollary from our observations is that code smells are just partial indicators of maintenance difficulties.

The explanatory power of code smells within source-code related difficulties was considerably low as well (58% of the cases), because some of the difficulties were associated to a combination of code smells and other characteristics in the code, whereas others were not associated code smells at all. The underlying reasons for difficulties that were associated to the presence of code smells relate back to already known notions such as size/complexity, and coupling. These characteristics are related to smells such as God Method, God Class, Feature Envy and ISP Violation.

However, we have also observed cases, where factors such as crosscutting concerns, and inconsistent design caused many of the difficulties. Based on our observations, we have identified individual smells and combinations that can help to recognize these code situations. Our observations also suggest that analyzing code smells individually may give the wrong picture, due to potential

interaction effects amongst code smells, and between smells and other code characteristics. We have discussed a critical case where the interaction effects between a design shortcoming and a code smell (ISP Violation) led to difficulties such as defects and costly changes. To achieve more comprehensive source-code evaluations, we suggest combining different code analysis techniques with code smell detection. In particular, we suggest the analysis of interaction effects amongst code smells via dependencies across classes and interfaces.

Future work includes better understanding inter-smell relations, their consequences for maintenance difficulties, and their effects on effort, defects and change size. We intend to study this aspect with focus on quantitative analysis, supported by qualitative observations. The results from this study are intended as starting points for developing hypotheses that can be tested via quantitative analysis in larger maintenance contexts. In addition, we plan to investigate more in detail potential "new" smells (as reported in Section 4.3.1), such as the "lack of conceptual integrity", "cyclic dependencies" and comprehension problems caused by "multiple inheritance and dynamic binding".

Appendix A. Excerpt of maintenance difficulties identified

Note that the selection of problems in Table A.15 does not reflect the actual distribution of code-related vs non-code related problems but was made so it includes the descriptions for all code-related problems mentioned in later tables (see Tables A.16, A.17 and A.18).

References

[1] K.H. Bennett, An introduction to software maintenance, Information and Software Technology (IST) 12 (1990) 257–264.
[2] W. Harrison, C. Cook, Insights on improving the maintenance process through software measurement, in: IEEE International Conference on Software Maintenance (ICSM), pp. 37–45.
[3] A. Abran, H. Nguyenkim, Analysis of maintenance work categories through measurement, in: IEEE International Conference on Software Maintenance (ICSM), pp. 104–113.
[4] T.M. Pigoski, Practical Software Maintenance: Best Practices for Managing Your Software Investment, Wiley, 1996.
[5] T.C. Jones, Estimating Software Costs, McGraw-Hill, 1998.
[6] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice, Springer, 2005.
[7] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[8] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, JDeodorant: Identification and removal of feature envy bad smells, in: IEEE International Conference on Software Maintenance (ICSM), pp. 519–520.
[9] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, JDeodorant: identification and removal of type-checking bad smells, in: 12th European Conference on Software Maintenance and Reengineering (CSMR), IEEE, 2008, pp. 329–331.
[10] C. Kiefer, A. Bernstein, J. Tappolet, Mining software repositories with iSPAROL and a software evolution ontology, in: Fourth International Workshop on Mining Software Repositories (MSR) (2007) 10.
[11] V.T. Rajlich, P. Gosavi, Incremental change in object-oriented programming, IEEE Software 21 (2004) 62–69.
[12] A.J. Riel, Object-Oriented Design Heuristics, first ed., Addison-Wesley, Boston, MA, USA, 1996.
[13] P. Coad, E. Yourdon, Object-Oriented Design, Prentice Hall, 1991.
[14] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.
[15] W. Brown, R. Malveau, S. McCormick, Tom Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, John Wiley & Sons Inc., 1998.
[16] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, third ed., Prentice Hall, 2004.
[17] R.C. Martin, Agile Software Development, Principles, Patterns and Practice, Prentice Hall, 2002.
[18] E. Van Emden, L. Moonen, Java quality assurance by detecting code smells, in: Working Conference on Reverse Engineering (WCRE), pp. 97–106.
[19] M.V. Mäntylä, J. Vanhanen, C. Lassenius, A taxonomy and an initial empirical study of bad smells in code, in: IEEE International Conference on Software Maintenance (ICSM), pp. 381–384.
[20] W.C. Wake, Refactoring Workbook, Addison-Wesley, 2003.
[21] G.H. Travassos, F. Shull, M. Fredericks, V.R. Basili, Detecting defects in object-oriented designs, in: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 47–56.
[22] M.V. Mäntylä, J. Vanhanen, C. Lassenius, Bad smells – humans as code critics, in: IEEE International Conference on Software Maintenance (ICSM), pp. 399–408.
[23] M.V. Mäntylä, C. Lassenius, Subjective evaluation of software evolvability using code smells: an empirical study, Empirical Software Engineering 11 (2006) 395–431.
[24] M.V. Mäntylä, An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement, in: International Conference on Software Engineering (ICSE), pp. 277–286.
[25] R. Marinescu, D. Ratiu, Quantifying the quality of object-oriented design: the factor-strategy model, in: Working Conference on Reverse Engineering (WCRE), IEEE, 2004, pp. 192–201.
[26] R. Marinescu, Measurement and quality in object-oriented design, in: IEEE International Conference on Software Maintenance (ICSM), pp. 701–704.
[27] N. Moha, Y.-g. Gueheneuc, P. Leduc, Automatic generation of detection algorithms for design defects, in: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2006, pp. 297–300.
[28] N. Moha, Detection and correction of design defects in object-oriented designs, in: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 949–950.
[29] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, L. Duchien, A domain analysis to specify design defects and generate detection algorithms, in: Fundamental Approaches to Software Engineering (FASE), pp. 276–291.
[30] A.A. Rao, K.N. Reddy, Detecting bad smells in object oriented design using design change propagation probability matrix, in: International MultiConference of Engineers and Computer Scientists, pp. 1001–1007.
[31] E.H. Alikacem, H.A. Sahraoui, A metric extraction framework based on a high-level description language, in: IEEE International Conference on Source Code Analysis and Manipulation (SCAM), pp. 159–167.
[32] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. Le Meur, DECOR: a method for the specification and detection of code and design smells, IEEE Transactions on Software Engineering 36 (2010) 20–36.
[33] M. Zhang, T. Hall, N. Baddoo, Code bad smells: a review of current knowledge, Journal of Software Maintenance and Evolution: Research and Practice 23 (2011) 179–202.
[34] A. Monden, D. Nakae, T. Kamiya, S. Sato, K. Matsumoto, Software quality analysis by code clones in industrial legacy software, in: IEEE Symposium on Software Metrics, pp. 87–94.
[35] W. Li, R. Shatnawi, An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution, Journal of Systems and Software 80 (2007) 1120–1128.
[36] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, Do code clones matter? in: International Conference on Software Engineering (ICSE), pp. 485–495.
[37] M. D'Ambros, A. Bacchelli, M. Lanza, On the impact of design flaws on software defects, in: International Conference on Quality Software (QSIC), pp. 23–31.
[38] F. Rahman, C. Bird, P. Devanbu, Clones: what is that smell? in: Working Conference on Mining Software Repositories (MSR), pp. 72–81.
[39] I. Deligiannis, M. Shepperd, M. Roumeliotis, I. Stamelos, An empirical investigation of an object-oriented design heuristic for maintainability, Journal of Systems and Software 65 (2003) 127–139.
[40] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, M. Shepperd, A controlled experiment investigation of an object-oriented design heuristic for maintainability, Journal of Systems and Software 72 (2004) 129–143.
[41] A. Lozano, M. Wermelinger, Assessing the effect of clones on changeability, in: IEEE International Conference on Software Maintenance (ICSM), pp. 227–236.
[42] M. Abbes, F. Khomh, Y.-G. Gueheneuc, G. Antoniol, An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, in: 15th European Conference on Software Maintenance and Reengineering (CSMR), IEEE, 2011, pp. 181–190.
[43] M. Kim, V. Sazawal, D. Notkin, G.C. Murphy, An empirical study of code clone genealogies, in: Joint 10th European Software Engineering Conference (ESEC) and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13), pp. 187–196.
[44] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, An exploratory study of the impact of code smells on software change-proneness, in: Working Conference on Reverse Engineering (WCRE), IEEE, 2009, pp. 75–84.
[45] S.M. Olbrich, D.S. Cruzes, D.I.K. Sjøberg, Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems, in: IEEE International Conference on Software Maintenance (ICSM), pp. 1–10.
[46] N. Tsantalis, A. Chatzigeorgiou, Identification of extract method refactoring opportunities for the decomposition of methods, Journal of Systems and Software 84 (2011) 1757–1782.
[47] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Transactions on Software Engineering 35 (2009) 347–367.
[48] Y. Higo, S. Kusumoto, K. Inoue, A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system, Journal of Software Maintenance and Evolution: Research and Practice 20 (2008) 435–461.
[49] G. Bavota, A. De Lucia, R. Oliveto, Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures, Journal of Systems and Software 84 (2011) 397–414.
[50] M. Fokaefs, N. Tsantalis, E. Stroulia, A. Chatzigeorgiou, Identification and application of Extract Class refactorings in object-oriented systems, Journal of Systems and Software 85 (2012) 2241–2260.
[51] K. Hotta, Y. Higo, S. Kusumoto, Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph,

in: European Conference on Software Maintenance and Reengineering, IEEE, pp. 53–62.
[52] N. Tsantalis, A. Chatzigeorgiou, Identification of refactoring opportunities introducing polymorphism, Journal of Systems and Software 83 (2010) 391–404.
[53] H. Liu, Z. Niu, Z. Ma, W. Shao, Identification of generalization refactoring opportunities, Automated Software Engineering 20 (2012) 81–110.
[54] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of bad smells in object-oriented code, in: International Conference on Quality of Information and Communications Technology, IEEE, 2010, pp. 106–115.
[55] R. Peters, A. Zaidman, Evaluating the lifespan of code smells using software repository mining, in: European Conference on Software Maintenance and Reengineering, IEEE, 2012, pp. 411–416.
[56] R. Yin, Case Study Research: Design and Methods (Applied Social Research Methods), SAGE, 2002.
[57] B.A. Kitchenham, G.H. Travassos, A. von Mayrhauser, F. Niessink, N.F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, H. Yang, Towards an ontology of software maintenance, Journal of Software Maintenance: Research and Practice 11 (1999) 365–389.
[58] B.P. Lientz, E.B. Swanson, Problems in application software maintenance, Communications of the ACM 24 (1981) 763–769.
[59] S. Dekleva, Delphi study of software maintenance problems, in: IEEE International Conference on Software Maintenance (ICSM), pp. 10–17.
[60] P. Palvia, A. Patula, J. Nosek, Problems and issues in application software maintenance, Journal of Information Technology Management 4 (1995) 17–28.
[61] N. Chapin, J.E. Hale, K.M. Kham, J.F. Ramil, W.-G. Tan, Types of software evolution and software maintenance, Journal of Software Maintenance: Research and Practice 13 (2001) 3–30.
[62] T. Hall, A. Rainer, N. Baddoo, S. Beecham, An empirical study of maintenance issues within process improvement programmes in the software industry, in: IEEE International Conference on Software Maintenance (ICSM), pp. 422–430.
[63] J.-C. Chen, S.-J. Huang, An empirical analysis of the impact of software development problem factors on software maintainability, Journal of Systems and Software 82 (2009) 981–992.
[64] A. Reedy, D. Stephenson, E. Dudar, F. Blumberg, Software configuration management issues in the maintenance of Ada software systems, in: IEEE International Conference on Software Maintenance (ICSM), pp. 234–245.
[65] A. Karahasanović, A.K. Levine, R. Thomas, Comprehension strategies and difficulties in maintaining object-oriented systems: an explorative study, Journal of Systems and Software 80 (2007) 1541–1559.
[66] A. Karahasanović, R.C. Thomas, Difficulties experienced by students in maintaining object-oriented systems: an empirical study, in: Australasian Conf. on Computing Education (ACE), Australian Computer Society, 2007, pp. 81–87.
[67] A. von Mayrhauser, A.M. Vans, Industrial experience with an integrated code comprehension model, Software Engineering Journal 10 (1995) 171–182.
[68] K. Webster, K.M. de Oliveira, N. Anquetil, A risk taxonomy proposal for software maintenance, in: IEEE International Conference on Software Maintenance (ICSM), pp. 453–461.
[69] R. Charette, K. Adams, M. White, Managing risk in software maintenance, IEEE Software 14 (1997) 43–50.
[70] S.L. Pfleeger, Software Engineering: Theory and Practice, Prentice Hall, 2001.
[71] N.F. Schneidewind, Requirements risk and maintainability, in: Advances in Software Maintenance Management: Technologies and Solutions, IGI Global, 2003, pp. 182–200.
[72] A. Yamashita, L. Moonen, Do code smells reflect important maintainability aspects? in: IEEE International Conference on Software Maintenance (ICSM), pp. 306–315.
[73] B.C.D. Anda, D.I.K. Sjøberg, A. Mockus, Variability and reproducibility in software engineering: a study of four companies that developed the same system, IEEE Transactions on Software Engineering 35 (2009) 407–429.
[74] G.R. Bergersen, J.-E. Gustafsson, Programming skill, knowledge, and working memory among professional software developers from an investment theory perspective, Journal of Individual Differences 32 (2011) 201–209.
[75] L.M. Layman, L.A. Williams, R.St. Amant, MimEc, in: International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), ACM Press, New York, USA, 2008, pp. 73–76.
[76] R. Marinescu, Measurement and Quality in Object Oriented Design, Doctoral thesis, Politehnica University of Timisoara, 2002.
[77] C. Kapser, M. Godfrey, "Cloning considered harmful" considered harmful: patterns of cloning in software, Empirical Software Engineering 13 (2008) 645–692.
[78] W. Wang, M.W. Godfrey, A study of cloning in the Linux SCSI drivers, in: IEEE 11th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2011, pp. 95–104.
[79] W. Stevens, G. Myers, Structured design, IBM Systems Journal 13 (1974) 115–139.
[80] J.I. Maletic, A. Marcus, Supporting program comprehension using semantic and structural information, in: International Conference on Software Engineering (ICSE), ICSE '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 103–112.
[81] A. Sharma, P.S. Grover, R. Kumar, Dependency analysis for component-based software systems, ACM SIGSOFT Software Engineering Notes 34 (2009) 1.
[82] E. Tempero, R. Biddle, Simulating multiple inheritance in Java, Journal of Systems and Software 55 (2000) 87–100.
[83] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler, Making the future safe for the past, in: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, New York, USA, 1998, pp. 183–200.
[84] E. Arisholm, D.I.K. Sjøberg, Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software, IEEE Transactions on Software Engineering 30 (2004) 521–534.
[85] R.E. Wood, Task complexity: definition of the construct, Organizational Behavior and Human Decision Processes 37 (1986) 60–82.
[86] B. Pietrzak, B. Walter, Leveraging code smell detection with inter-smell relations, in: Extreme Programming and Agile Processes in Software Engineering (XP), pp. 75–84.
[87] B.C.D. Anda, Assessing software system maintainability using structural measures and expert assessments, in: IEEE International Conference on Software Maintenance (ICSM), pp. 204–213.
[88] B. Walter, B. Pietrzak, 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, 2005, pp. 154–161.
[89] J.A. Maxwell, Using qualitative methods for causal explanation, Field Methods 16 (2004) 243–264.
[90] H. Becker, B. Geer, Participant observation and interviewing: a comparison, Human Organization 16 (1957) 28–32.
[91] Y. Guo, C. Seaman, N. Zazworka, F. Shull, Domain-specific tailoring of code smells: An empirical study, in: International Conference on Software Engineering (ICSE), vol. 2, pp. 167–170.