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