Information and Software Technology 96 (2018) 112–125

Contents lists available at ScienceDirect

Information and Software Technology

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

An empirical study to improve software security through the application of code refactoring

Haris Mumtaz, Mohammad Alshayeb*, Sajjad Mahmood, Mahmood Niazi

Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia

A R T I C L E I N F O

Keywords:
Refactoring
Secured software
Empirical study
Code bad smells

A B S T R A C T

Context: Code bad smells indicate design flaws that can degrade the quality of software and can potentially lead to the introduction of faults. They can be eradicated by applying refactoring techniques. Code bad smells that impact the security perspective of software should be detected and removed from their code base. However, the existing literature is insufficient to support this claim and there are few studies that empirically investigate bad smells and refactoring opportunities from a security perspective.
Objective: In this paper, we investigate how refactoring can improve the security of an application by removing code bad smell.
Method: We analyzed three different code bad smells in five software systems. First, the identified code bad smells are filtered against security attributes. Next, the object-oriented design and security metrics are calculated for the five investigated systems. Later, refactoring is applied to remove security-related code bad smells. The correctness of detection and refactoring of investigated code smells are then validated. Finally, both traditional object-oriented and security metrics are again calculated after removing bad smells to assess its impact on the design and security attributes of systems.
Results: We found 'feature envy' to be the most abundant security bad smell in investigated projects. The 'move method' and 'move field' are commonly applied refactoring techniques because of the abundance of feature envy.
Conclusion: The results of security metrics indicate that refactoring helps improve the security of an application without compromising the overall quality of software systems.

1. Introduction

Code bad smells indicate poor coding and design choices that can directly or indirectly lead to the introduction of faults [1–3]. Code bad smells impact the structural characteristics of software such that they contribute to degrading the quality of the software [1,2]. Refactoring is a widely used technique to improve the quality of software while preserving the functionalities and behavior [2]. Several studies have been proposed for bad smell detection and subsequent refactoring to improve code quality aspects such as understandability, changeability and overall maintenance [4]. Bavota et al. investigated the bad test code smells [5].

Security is an important software quality aspect that reflects the ability of a system to prevent data exposure and loss of information. A basic aim of secure software is to prevent unauthorized access and modification of information. The fulfillment of security requirements at the design and implementation level is imperative to minify the cost of addressing this issue at later stages of development and maintenance. Similar to other quality attributes, it is important to evaluate how code bad smells affect the security of software. Furthermore, there is a need to investigate whether refactoring techniques help improve the security of software by removing code bad smells. Security itself is a wide domain and its applicability is varied and context dependent. In the current study context, we investigate whether source code fragments face security issues and if they violate security requirements. The confinement of our study to source code has allowed us to infer the security requirements in a manner to show their applicability to source code. The detected bad smells in our study are rigorously analyzed against security requirements to deduce with certainty about the effect of bad smells from a security perspective. In our study, the bad smells violating security requirements are referred to as "security bad smells". Previous studies have rigorously examined the relations between individual code bad smells and quality attributes such as maintenance effort and defect prediction [4]; yet no study has investigated how refactoring can improve software security by detecting and removing code bad smells.

This paper empirically investigates how refactoring can improve the

* Corresponding author.
E-mail addresses: g201405820@kfupm.edu.sa (H. Mumtaz), alshayeb@kfupm.edu.sa (M. Alshayeb), smahmood@kfupm.edu.sa (S. Mahmood), mkniazi@kfupm.edu.sa (M. Niazi).

https://doi.org/10.1016/j.infsof.2017.11.010
Received 22 November 2016; Received in revised form 13 August 2017; Accepted 18 November 2017
Available online 22 November 2017

security of an application by removing code bad smells. The study follows the guidelines provided by Jedlitschka et al. [6]. We report on four open source and one student projects. We used *inFusion* [7], a well-known bad smell detection tool [8–14], that has the ability to detect more than 20 design flaws and code smells. In this paper, we used inFusion for its ability to detect the investigated bad smells. The identified code bad smells are automatically removed using the *IntelliJIDEA* tool. Subsequently, security metrics data is used to determine the presence of security concerns in the analyzed projects and to assess the impact of refactoring on the design and security attributes.

The rest of the paper is structured as follows: Section 2 provides a preliminary understanding of security characteristics and the code refactoring process. Section 3 provides a detailed description of the related work in terms of code bad smells, security aspects of software development, metrics-based bad smell detection and refactoring techniques. Section 4 specifies the research question based on the problem statement identified from the gaps in the current literature. It also highlights the basic components of our research methodology. Section 5 provides details on the experimental design. The analysis and discussion on the obtained results are presented in Section 6. Section 7 identifies the threats to validity of this research and finally, Section 8 concludes the paper and provides the future research directions.

2. Background

This section provides a discussion on security characteristics. It also illustrates the refactoring process applied to deal with code bad smells. A brief discussion on existing refactoring tools is also covered in this section.

2.1. Security characteristics

A variety of software quality characteristics has been reported in the literature such as: performance, scalability, modifiability, security, availability, integration, portability and testability [15]. The measurement strategies have also been studied for these quality attributes in object-oriented code [16–18]. One of the renowned methods to measure software quality attributes is object-oriented metrics analysis. Security is one of the important quality attributes in software systems. The most common factors to measure security are: confidentiality, integrity and availability [15,19–21]. These security measurement factors, however, are subjective in nature and trivial to quantify.

According to Whitman and Mattord [21], information security is to protect the storage, processing and exchange of information from confidentiality, integrity and availability perspectives. These three perspectives are major components of ensuring security. When information is protected from unauthorized access, it means confidentiality is ensured [21]. The integrity of information is compromised when information is exposed to damage, corruption or any kind of disruption [21]. This security metric affects consistency, completeness and correctness attributes of software quality [22]. The third major security attribute is availability, which means that data and services are available to authorized users at all times [21]. Jürjens listed some additional security characteristics including: fair exchange, non-repudiation, role-based access control, secure communication links, secrecy and integrity, authenticity, freshness, secure information flow and guarded access [19]. Gorton also reported some requirements that a software system should encapsulate to ensure security. The identified requirements include: authentication, authorization, encryption, integrity and non-repudiation [15].

2.2. Code refactoring process and tools

Software refactoring means that software design or code is transformed in such a way that it improves software quality while preserving its behavior [2]. Opdyke introduced the refactoring concept and proposed several refactoring opportunities at both the design and implementation level [23].

Refactoring is usually conducted as a standard process for model and source code, comprising several steps. The process was initially presented by Wake [24], and further extended by Mens and Tourwé [25]. In general, the code refactoring process comprises the following steps:

Step 1: Identifying software parts that require refactoring.
Step 2: Selecting the appropriate refactoring approach.
Step 3: Checking for behavioral preservation.
Step 4: Applying the selected refactoring approach.
Step 5: Analyzing the impact of refactoring on software quality improvement.
Step 6: Ensuring consistency between the refactored code and the corresponding UML class model.

The details related to steps 1 and 2 are provided in Section 3.3. The discussion related to metrics-based detection techniques basically describes different ways of capturing portions of software which require refactoring. The third step focuses on code behavioral preservation before it is exposed to refactoring. One way of achieving behavioral preservation is by defining the pre- and post-conditions of refactoring [23,26]. Opdyke [23] used the assistance of preconditions to ensure behavioral preservation in code that is targeted to be refactored. Preconditions basically ensure that refactored code preserves its behavior. But it is achieved with additional overhead cost to the refactoring process. Roberts [26] intended to verify behavioral preservation with the use of post-conditions. The reason for considering post-conditions instead of preconditions was the belief in delaying the refactoring process, because undertaking the verification activity before refactoring would certainly delay the refactoring activity.

The application of code refactoring can be achieved manually, in a semi-automated manner or a fully-automated manner. Many code refactoring tools are available and are listed in Table 1 with the available refactoring strategies.

The next stage of the refactoring process is the evaluation of corrected code to assess the improvement in quality. Although evaluation of refactored code is an imperative activity, unfortunately only a few studies have actually focused on it [30]. The researchers mainly use software metrics to make judgments on quality improvement [31–33]. The metrics values are calculated before and after refactoring to assess the impact of refactoring on code quality. Enckevort [31] used Fan-in, Fan-out and CK metrics [34] to quantify code quality. Moghadam and

Table 1
Refactoring tools.

Tool	Refactoring strategies
InterlliJ IDEA [27]	Rename and Move Program Entities, Change Method Signature, Extract Method, Inline Method, Introduce Variable, Introduce Field, Inline Local Variable, Extract Interface, Extract Superclass, Encapsulate Fields, Pull Up Members, Push Down Members and Replace Inheritance with Delegation.
RefactorIt [28]	Rename, Move Class, Move Method, Encapsulate Field, Create Factory Method, Extract Method, Extract Superclass/Interface, Minimize Access Rights, Clean Imports, Create Constructor and Pull Up/Push Down Members.
JRefactory [29]	Move Class, Rename Class, Add an Abstract Superclass, Remove Class, Push Up Field, Pull Down Field and Move Method.