


The Study on Software Architecture Smell Refactoring

Kuo Jong-Yih, National Taipei University of Technology, Taiwan

Hsieh Ti-Feng, National Taipei University of Technology, Taiwan

Lin Yu-De, National Taipei University of Technology, Taiwan

Lin Hui-Chi, National Taipei University of Technology, Taiwan*

 <https://orcid.org/0000-0002-0492-5428>

ABSTRACT

Maintenance and complexity issues in software development continue to increase because of new requirements and software evolution, and refactoring is required to help software adapt to the changes. The goal of refactoring is to fix smells in the system. Fixing architectural smells requires more effort than other smells because it is tangled in multiple components in the system. Architecture smells refer to commonly used architectural decisions that negatively impact system quality. They cause high software coupling, create complications when developing new requirements, and are hard to test and reuse. This paper presented a tool to analyze the causes of architectural smells such as cyclic dependency and unstable dependency and included a priority metric that could be used to optimize the smell with the most refactoring efforts and simulate the most cost-effective refactoring path sequence for a developer to follow. Using a real case scenario, a refactoring path was evaluated with real refactoring execution, and the validity of the path was verified.

KEYWORDS

Architecture Smell, Refactoring Strategies, Refactoring Tool

INTRODUCTION

In the software development life cycle (SDLC), the scale of a software project will grow because of the evolution in software requirements, IT equipment upgrades, and technology change (Lehman et al., 1996), which cause the cost of software maintenance and its complexity to increase. In order to maintain the quality of a project, teams will need to perform code refactoring regularly to reduce the accumulation of project technical debt (Suryanarayana et al., 2014). The best chance to do refactoring in a project is the region where smells are located. The smell is a surface indication that usually corresponds to a deeper problem in the system (Fowler et al., 1999). It can be classified into code smell (Fowler et al., 1999), design smell (Suryanarayana et al., 2014), and architectural smell (Lippert et al., 2006).

DOI: 10.4018/IJSI.339884

*Corresponding Author

This article published as an Open Access article distributed under the terms of the Creative Commons Attribution License (<http://creativecommons.org/licenses/by/4.0/>) which permits unrestricted use, distribution, and production in any medium, provided the author of the original work and original publication source are properly credited.

Architecture smell (AS) is defined as common, but not always intentional, solutions used in architectural decisions that negatively impact software quality (Garcia et al., 2009). AS has relations with software architecture, and it may be involved in the division of a system into components, the arrangement of those components, and the ways in which those components communicate with each other (Martin, 2017).

The refactoring of AS involves coordinating a set of deliberate architectural activities that remove a particular architectural smell and improve at least one quality attribute without changing the system's scope and functionality (Sas et al., 2019). To help developers to remove AS, we developed a tool prototype as a support for AS refactoring that could analyze the actual cause of the AS and the recommended refactoring process based on the architecture smell using variable parameters and characteristic metrics (Arcelli et al., 2017).

The remainder of this paper is structured as follows: the second section introduces relevant terms in the field of architectural smells (AS), architectural smell refactoring, and related tools. The third section presents the research methodology used in this study and outlines the design of the refactoring process strategies. The fourth section describes the implementation of the device, presents a case study, and analyzes the results. Section five serves as the conclusion of the research.

RELATED WORK

Architecture Smell

Architecture smell is considered to violate the common design principle and affects the internal quality of software. It increases the coupling of components and may break the modularity of the system. Different authors have provided different definitions of AS according to different levels, such as Lippert et al., (2006), who defined AS's in dependency graphs, packages, subsystems, layers, and so on. Fontana et al., (2019) propose a tool called Arcan developed for the detection of architectural smells. Evaluate the PageRank and Criticality of these smells through the analysis of six projects These architectural smells are categorized into three types based on dependency issues, such as cyclic dependency (CD), unstable dependency (UD), and hub-like dependency (HL). This analysis has provided the architecture smell related to dependency issues, such as cyclic dependency (CD), unstable dependency (UD), and hub-like dependency (HL). Azadi et al., (2019) provide a proposal for AS classification (Figure 1) based on the violation of three design principles, including the principles of modularity (Suryanarayana et al., 2014), hierarchy (Suryanarayana et al., 2014), and healthy dependency structure (Caracciolo et al., 2016).

The AS chosen in this study included CD and UD, which can be detected by the Arcan tool in the detection of three smells in two industrial projects (Arcelli et al., 2016), and both violate the principle of the healthy dependency structure. CD also violates the principle of modularity, making it difficult to modify the requirements in the system and affecting the changeability and reusability of components related to the AS.

Cyclic Dependency

Cyclic Dependency (CD) represents a cycle among several components; it will lead the side effect when we try to modify the components in cycle. There are several software design principles that suggest avoiding creating such cycles, like Acyclic Dependencies Principle (Martin, 2003) and The Common Closure Principle (Robert, 2003). CD may have different topological shapes, which is shown in Figure 2, provided by Al-Mutawa et al., (2014). More complex shapes mean that the maintainability of the system is reduced because of the affected part.

15 more pages are available in the full version of this document, which may be purchased using the "Add to Cart" button on the publisher's webpage: www.igi-global.com/article/the-study-on-software-architecture-smell-refactoring/339884

Related Content

State of Practice in Secure Software: Experts' Views on Best Ways Ahead

Bill Whyte and John Harrison (2011). *Software Engineering for Secure Systems: Industrial and Research Perspectives* (pp. 1-14).

www.irma-international.org/chapter/state-practice-secure-software/48404

Test Case Prioritization using Cuckoo Search

Praveen Ranjan Srivastava, D. V. Pavan Kumar Reddy, M. Srikanth Reddy, Ch. V. B. Ramaraju and I. Ch. Manikanta Nath (2012). *Advanced Automated Software Testing: Frameworks for Refined Practice* (pp. 113-128).

www.irma-international.org/chapter/test-case-prioritization-using-cuckoo/62153

Exploiting Codified User Task Knowledge to Discover Services at Design-Time

Konstantinos Zachos, Angela Kounkou and Neil A. M. Maiden (2012). *International Journal of Systems and Service-Oriented Engineering* (pp. 30-66).

www.irma-international.org/article/exploiting-codified-user-task-knowledge-to-discover-services-at-design-time/78917

Design Space Exploration for Implementing a Software-Based Speculative Memory System

Kohei Fujisawa, Atsushi Nunome, Kiyoshi Shibayama and Hiroaki Hirata (2018). *International Journal of Software Innovation* (pp. 37-49).

www.irma-international.org/article/design-space-exploration-for-implementing-a-software-based-speculative-memory-system/201484

A Case Study on Testing for Software Security: Static Code Analysis of a File Reader Program Developed in Java

Natarajan Meghanathan and Alexander Roy Geoghegan (2012). *Advanced Automated Software Testing: Frameworks for Refined Practice* (pp. 89-112).

www.irma-international.org/chapter/case-study-testing-software-security/62152