The Journal of Systems and Software 107 (2015) 142–157

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

An automated approach for noise identification to assist software architecture recovery techniques

Eleni Constantinou [a,*], George Kakarontzas [a,b], Ioannis Stamelos [a]

[a] Department of Informatics, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece
[b] Computer Science and Engineering Department, Technological Educational Institute of Thessaly, 41110 Larissa, Greece

ARTICLE INFO

Article history:
Received 11 September 2014
Revised 28 April 2015
Accepted 27 May 2015
Available online 5 June 2015

Keywords:
Noise identification
Omnipresent classes
Software architecture recovery

ABSTRACT

Software systems' concrete architecture often drifts from the intended architecture throughout their evolution. Program comprehension activities, like software architecture recovery, become very demanding, especially for large and complex systems due to the existence of noise, which is created by omnipresent and utility classes that obscure the system structure. Omnipresent classes represent crosscutting concerns, utilities or elementary domain concepts. The identification and filtering of noise is a necessary preprocessing step before attempting program comprehension techniques, especially for undocumented systems. In this paper, we propose an automated methodology for noise identification. Our methodology is based on the notion that noisy classes are widely used in a system, directly or indirectly. We combine classes' usage significance with their participation in the system's subgraphs, in order to identify the classes that are persistently used. Usage significance is measured according to Component Rank, a well-established metric in the literature, which ranks software artifacts according to their usage significance. The experimental results show that the proposed methodology successfully captures classes that produce noise and improves the results of existing algorithms for software systems' architectural decomposition.

1. Introduction

Software systems' intended architecture deviates from the implemented architecture throughout their evolution over time (Ducasse and Pollet, 2009). Additionally, architectural or implementation documentation is often outdated or missing in several important categories of software systems, e.g. legacy systems, open source software systems, etc (Shtern and Tzerpos, 2012). Program comprehension techniques, like Software Architecture Reconstruction (SAR), mainly draw information about software systems from human experts and source code artifacts (Pollet et al., 2007). However, human expertise is not always available since the comprehension of large and complex software systems lacking adequate documentation is time consuming (Chardigny and Seriai, 2010). Source code artifacts analysis is predominant in program comprehension techniques, where the concrete architecture derived from the source code is utilized to gain knowledge of the intended or conceptual architecture of a software system (Pollet et al., 2007; Maqbool and Babri, 2007).

Program comprehension techniques require a preprocessing step to filter out noise (Maqbool and Babri, 2007). Noise in software systems is produced by classes that are intensively utilized either system-wise, i.e. omnipresent classes, or by their local neighborhood. Omnipresent classes represent crosscutting concerns, utility functionalities or elementary domain concepts such as entities. Their basic attributes are that they are called by a vast number of classes in the system (Maqbool and Babri, 2007; Mancoridis et al., 1999; Mitchell and Mancoridis, 2006; Muller and Uhl, 1990; Muller et al., 1993; Luo et al., 2005; Wen and Tzerpos, 2005; Zhang et al., 2010), directly or indirectly, and they are usually located in the bottom architectural layers. However, they are not very important from an architectural point of view in SAR activities, since they do not necessarily represent architecturally significant decisions (Lungu et al., 2014). Although entity classes are significant for comprehending software systems, in SAR they can produce noise in the system structure since they are used by a significant proportion of system's classes. Intensively used classes do not positively contribute to the clustering methodologies of SAR (Maqbool and Babri, 2007), because their pervasive presence in the systems induces considerable noise. Consequently, identifying and removing classes that produce noise in a software system is an important preparatory procedure before attempting the comprehension of architecturally significant elements, since their presence increases the gap between implementation and design (Pirzadeh et al., 2009). Additionally, the authors of a recent feature location taxonomy and survey (Dit et al., 2013) observe that setting up benchmark

* Corresponding author.
E-mail addresses: econst@csd.auth.gr (E. Constantinou), gkakaron@teilar.gr (G. Kakarontzas), stamelos@csd.auth.gr (I. Stamelos).

http://dx.doi.org/10.1016/j.jss.2015.05.065
0164-1212/

systems that can be used in several research papers is challenging, partially because benchmarks may not be 100% accurate since they might contain noise (i.e. program elements not related to the feature). One of our contributions is that we can assist in the exclusion of noise as a preparatory procedure prior to modularization or feature location processes and not only to SAR techniques. In general, any approach attempting to comprehend a system for the identification of higher-level constructs, such as architecturally significant program elements or program elements related to a feature, should first remove elements that represent noise in the system (Maqbool and Babri, 2007; Muller and Uhl, 1990; Muller et al., 1993).

In this paper we propose an automated methodology to identify classes producing noise to the structure of software systems. Our methodology initially performs static analysis to measure classes' usage significance. The metric used to measure classes' significance is the Component Rank (CR) model, presented by Inoue et al. (2005). CR model is based on an algorithm that iteratively recomputes classes' weights according to the use relationships among them. In a following step, we find the shortest paths between classes' pairs and utilize the usage significance values of the participant nodes in order to quantify the overall usage of each class in the system. The obtained weights for the system's paths are used to measure classes' significance in the graph according to their usage and finally, identify noise classes. We evaluate the proposed methodology on three Java applications, where we validate the structural attributes of Noise classes and compare the results with related approaches of the literature, i.e. omnipresent detection of the Bunch tool (Mancoridis et al., 1999; Mitchell and Mancoridis, 2006) and the approach of Zhang et al. (2010), based on which we further discuss our findings.

The contributions of our methodology are as follows:

- We introduce a fully automated methodology and therefore we minimize the effort required by software engineers for the detection of classes that cause noise in the system structure.
- We improve the applicability of the noise detection approaches by not requiring from software engineers to define the threshold value.
- We consider the usage intensity of a class in the system, which does not only depend on the number of incident edges, but is a combination of the direct and indirect incoming dependencies with the weights accumulated to them.

The rest of the paper is organized as follows. In Section 2 we present related work and in Section 3 the proposed methodology is described in detail. Then, in Section 4 we present the experimental results and in Section 5 we show the impact of the proposed approach to SAR techniques. In Section 6 we discuss our findings and in Section 7 we present threats to validity. Finally, in Section 8 we conclude our main contributions and provide future research directions.

2. Related work

Omnipresent classes were introduced by Muller and Uhl (1990) as modules referenced by most of the system components. The authors consider as omnipresent classes those nodes in the system graph that the number of their direct clients exceeds a threshold T_{op}. They argue that omnipresent components should be removed with their incident edges, since they obscure the system structure. In a following work (Muller et al., 1993), the authors suggest that further inspection of the items identified as omnipresent is required, since some central components of the system are also identified as omnipresent.

Mancoridis et al. (1999) cluster software systems to create the system decomposition and as a preprocessing step they identify omnipresent modules. They divide omnipresent classes in two categories, direct suppliers and clients, where in the former case the classes have a vast number of outgoing dependencies and in the latter case, of incoming dependencies. In order to identify omnipresent modules with their tool Bunch they specify a threshold, i.e. a multiple of the average edge in-degree and out-degree of the Module Dependency Graph (MDG), and automatically create the direct suppliers and clients lists with classes above the threshold. In a following work (Mitchell and Mancoridis, 2006), they discuss that omnipresent modules bias the clustering results in a negative way and therefore exclude them to simplify the graph before applying clustering.

Luo et al. (2005) initially identify omnipresent clusters in order to remove noise and perform hierarchical system decomposition. They distinguish three types of omnipresent classes: control, common and their combination according to their in-degree and out-degree. They specify a threshold and classes whose in-degree or out-degree is above the threshold are deemed as omnipresent. Next, they form clusters according to the class type by including neighbor classes whose minimal path from/to the omnipresent class is under a user specified threshold k. Finally, they produce omnipresent clusters by selecting candidates from clusters according to the maximal value of the Independency Metric (IM). Zhang et al. (2010) follow a similar approach to identify and remove omnipresent classes in order to improve the performance of system decomposition. However, they propose a modification for IM to reduce the impact of the sub-graph size on the IM value.

Tzerpos and Holt (2000) present a pattern based algorithm, namely ACDC, to decompose a software system into meaningful partitions. In the context of their work, they introduce several subsystem patterns for the ACDC algorithm. They present the support library pattern to discover and group the procedures that are accessed by the majority of the subsystems. They identify classes as omnipresent when their in-degree is larger than a predefined threshold, i.e. 20. However, this approach can produce misleading results in cases of large systems. Wen and Tzerpos (2005) present a set of detection methods for omnipresent classes by taking into account the subsystem structure when deciding if a class is omnipresent or not. They define the cluster degree d of classes as the number of clusters that this class is connected to, excluding its parent cluster. Finally, they introduce four detection methods, percentile, absolute and normalized cluster degree, and relative degree, related to literature methods. All detection methods require a threshold in order to conclude to omnipresent classes. However, the cluster degree calculation requires a clustered decomposition of the system; the authors point out that they applied their metrics on two systems of known authoritative decomposition. In contrast, our approach does not rely on existing decompositions, nor the results are prone to the decomposition used.

Further research related to noise detection involves the identification of utilities. Hamou-Lhadj et al. (2005) propose a technique to distinguish utility components from high-level concepts' components. They consider as utility any element of a program designed for the developers' convenience, which is intended to be accessed from multiple places. They introduce the utilityhood metric U, which is based on fan-in analysis, and present an algorithm to detect utility classes. Their utilityhood metric can be adjusted to detect utilities either system-wise (omnipresent classes) or for a particular package. For system utility identification, their metric is defined as the fraction of incoming connections, divided by the size of the system. Their algorithm initially computes U for each class and considers as utilities the classes with metric values greater than or equal to a predefined threshold, which is defined by software engineers. In a following work, they improve the utilityhood metric in order to depend both on fan-in and fan-out values (Hamou-Lhadj and Lethbridge, 2006). Although utilityhood metric aims towards utilities, the results include classes widely used in the system that can contain code related to system functionality (Dugerdil and Repond, 2010), i.e. a subset of omnipresent classes. Patel et al. (2009) decompose software systems and as a preprocessing step they remove omnipresent classes according to the approach of Hamou-Lhadj and Lethbridge (2006).