Information Processing Letters 115 (2015) 414–417

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Extracting reusable components: A semi-automated approach for complex structures

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

[a] Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki 54124, Greece
[b] Department of Computer Science and Telecom., T.E.I. of Larissa, Larissa 41110, Greece

ARTICLE INFO

Article history:
Received 1 July 2014
Received in revised form 2 November 2014
Accepted 17 November 2014
Available online 26 November 2014
Communicated by J.L. Fiadeiro

Keywords:
Software Engineering
Component Extraction
Cyclic Dependencies
Software Reuse

ABSTRACT

Source code comprehension depends on the source code quality and structural complexity. Software systems usually have complex structures with cyclic dependencies that make their comprehension very demanding. We present a semi-automated process that guides software engineers to untangle complex structures in order to extract reusable components. The process consists of iterative analysis in order to identify and transform the classes responsible for the structural complexity and effectively reducing candidate components' sizes. We evaluate our approach on two systems and demonstrate how the proposed approach assists the reusable component extraction.

1. Introduction

Reuse tasks of open source or legacy systems are facilitated when architectural and structural information is provided. However, architectural information often does not exist and thus, system understanding lies on the concrete architecture derived from the source code [1]. This can be a time consuming task since software systems do not comply with several design principles [2]. In particular, even though most design practices advise to comply with Acyclic Dependency Principle (ADP) [3], cyclic dependencies are widely observed since the most frequently deployed architecture is the Big Ball of Mud [4,5]. Melton and Tempero studied 78 open and closed source projects and report that 45% present cycles of 100 classes whereas 10% present cycles of 1000 classes [5].

Reuse approaches include the reusing isolated classes approach, where a small number of files should be copied in order to reduce the compilation time, the search space for classes' comprehension and the amount of code that could contain bugs [5]. However, a prerequisite is to extract all their dependencies to produce a compilable component. In cases of tangled structures with cyclic dependencies, this task can lead to the extraction of large components that contain classes not related to the required functionality. Overall, smaller components are easier to understand and adapt for reuse purposes. Therefore, we consider that components have manageable sizes when the software engineer can comprehend them without excessive effort.

The component extraction task starts with the selection of an origin class for the extraction of its corresponding component. The proposed approach guides software engineers to comprehend systems with complex structure and reduce the size of candidate components implicated in tangled dependencies. Thus, cyclic dependencies must be eliminated to facilitate source code reuse by applying the Dependency Inversion Principle (DIP). According to DIP, an interface of the class is introduced and the class

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

http://dx.doi.org/10.1016/j.ipl.2014.11.007
0020-0190/© 2014 Elsevier B.V. All rights reserved.

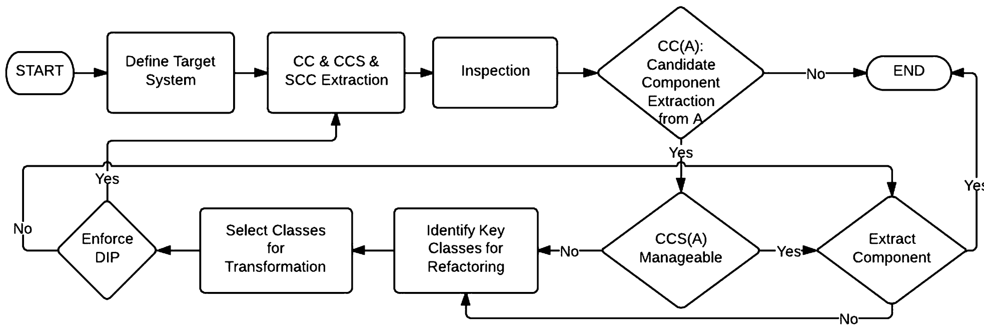
START
Define Target System
CC & CCS & SCC Extraction
Inspection
CC(A): Candidate Component Extraction from A
No
END
Yes
Yes
Yes
No
Enforce DIP
Select Classes for Transformation
Identify Key Classes for Refactoring
No
CCS(A) Manageable
Yes
Extract Component
No

Fig. 1. Component extraction process model.

implements the interface. Classes that were depending on this class now depend on the interface instead, and the class only depends on the interface. Thus, the dependency is inversed [3]. Finally, our approach iteratively analyzes the system in order to transform classes responsible for complex dependencies according to DIP.

The rest of the paper is organized as follows. In Sections 2 and 3 we present related work to component extraction and the proposed process respectively. In Section 4 we present and discuss the results, and in Section 5 we present the threats to validity. Finally, in Section 6 we conclude and provide future research directions.

2. Related work

Washizaki and Fukazawa [6] identify reusable parts of Java systems and transform them into reusable JavaBeans components. Although we consider a similar component extraction approach, our approach focuses on the reduction of candidate component sizes. Marx et al. [7] propose an approach to extract components starting from a set of predefined entities and iteratively transform them by refactorings with DIP in order to finalize the component. However, they evaluate their approach on small applications (72–139 classes), while our approach aims to larger and complex systems. Additionally, they only use DIP in order to isolate the component from other components, while we use DIP to transform the identified cut points in order to reduce the size of the component. Wang et al. [8] propose extracting components based on weighted connectivity strength (WCS) metrics and a hierarchical clustering algorithm. The main challenge they identify is that classes from different layers of the system are classified together due to their close relation, a barrier our approach attempts to overcome. Other works related to component extraction mainly focus on clustering systems into components [9–11]. However, such approaches do not consider an entry point for the component extraction and do not necessarily produce independent components. Therefore, the successful reuse of the extracted components is not guaranteed by these methods.

3. The proposed reusable component extraction process

A system's structure is modeled as a directed graph $G = \langle V, E \rangle$, where the set of nodes V and edges E represent the classes and the use relationships between them respectively. A use relationship includes all types of dependency between classes, i.e. inheritance, method call, etc. The component extraction is a task where the software engineer chooses the origin class that represents the entry point to the required provided functionality, and extracts the component. The candidate component (CC) $\mathrm{CC}(v_i)$ is the subgraph of G that is formed by including all the transitive dependencies of the origin class v_i. Candidate Component Size (CCS) is the size of the CC, $\mathrm{CCS}(v_i) = |\mathrm{CC}(v_i)|$. The CC set information is enriched by the level of dependency, that corresponds to the shortest path from the origin class to reach each dependency.

Initially, each class is considered as a CC and the corresponding CCS values are extracted. CCS explosion phenomenon occurs when the system's obtained CCS values initially present a linear increase, followed by a steep growth. This occurs due to complex cyclic structures that implicate a large number of classes. Complex dependencies include cycles of classes, since they are formed by sets of classes that depend directly or indirectly on each other and can contain subcycles. Cyclic dependencies are identified according to Tarjan's algorithm [12], where Strongly Connected Components (SCCs) are recovered. Cycles coincide with SCCs, since by definition SCCs are sets of vertices such that for each pair of vertices within the set, there is a directed path between them [12].

Fig. 1 presents the component extraction process. Initially, the software engineer defines the system under investigation and then, the system is analyzed to obtain information about the SCCs, CCs, and CCS. The analysis results are presented to the software engineer so as to inspect the attributes of the classes intended for component extraction (e.g., CCS). If no origin classes are identified, the process ends. Otherwise, he identifies the origin class and if its CCS value is manageable, the component is extracted and the process terminates.

If the CC suffers from the CCS explosion phenomenon the software engineer provides the origin class as input to the analysis and the outcome is a set of key classes (KC) that propagate the CCS explosion phenomenon. More specifically, key classes are members of the CC, $\mathrm{KC} \subset \mathrm{CC}$, that participate in cyclic dependencies, $\mathrm{KC} \in \mathrm{SCC}(Y)$, where Y is a cycle in graph G. Thus, they share the same CCS value regardless of their level of dependency to the origin class, $\mathrm{CCS}(x) = \mathrm{CCS}(y)\ \forall x, y \in \mathrm{KC}$. The key classes are identified by a top down search to each CC level of dependency of the origin class. Initially, classes with identical