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/

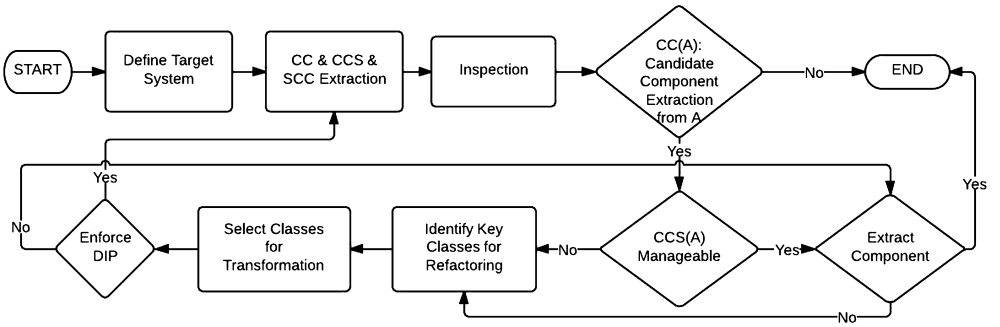

Fig. 1. Component extraction process model.

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

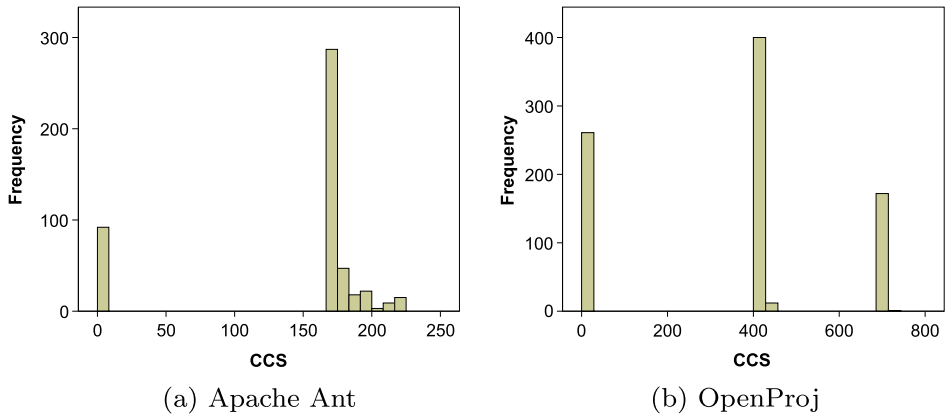

Fig. 2. CCS Values.

Table 1
Process application.

CCS	Ant			OpenProj	
	CC(*Equals*)	CC(*Contains*)	CC(*HasFreeSpace*)	CC(*Duration*)	CC(*Merge*)
Iteration 1	173	173	175	402	402
Iteration 2	127	127	129	16	23
Iteration 3	5	5	10		

size as the origin class are identified, $CCS(x) = CCS(origin)$ $\forall x \in KC$, that are unique in their level of dependency. These classes and their direct (i.e., level 1) dependencies with the same CCS value are imported to the candidate set in order not to omit possible candidates. Finally, key classes are ranked according to the largest number of direct dependencies in the candidate set, since the highest ranked classes contribute to the CCS explosion phenomenon.

The software engineer inspects the highest ranked classes and selects the classes to be transformed by the application of DIP. For example, a key class can implement a crosscutting concern of the system, e.g., logging, and by eliminating or transforming this class, the CC can be completely detached from any cyclic dependencies. The size of the resulting CC after the transformation is defined as $CCS' = \sum_{i=1}^{n} |CC_i|$, where n is the level of dependency of the selected class for transformation and CC_i are the classes in the level of dependency i of the CC. If he decides that system transformation is not required, he extracts the CC and the process is terminated. Alternatively, he indicates the classes to be transformed, the system is reanalyzed and a new snapshot of the system is produced. The aforementioned steps are repeated until a manageable CCS value is reached. The steps of the process are supported by ReCompId[1] tool.

4. Results

We present[2] the application of our process on two Java systems of different size and domain, i.e., Apache Ant (493 classes) and OpenProj (846 classes). Figs. 2a and 2b present the frequency that each CCS value is encountered if each class in the system is selected for component extraction, e.g., Ant contains 287 CCs with size 172–174. Both systems suffer from CCS explosion due to the existence of cycles that implicate large proportion of systems' CCs.

Ant contains two cycles responsible for large CCS values, where CCS values equal to 172 and 215 for each cycle. The process is applied to three CCs that suffer from CCS explosion, namely CC(*Equals*), CC(*Contains*), and CC(*HasFreeSpace*). *Equals* and *Contains* provide String evaluation functionalities and their CCS values equal 173. *HasFreeSpace* functionality includes hard disk related operations and its CCS value equals 175. Initially, the software engineer triggers the process for CC(*Equals*). In the first iteration, *Path* class is ranked first so as to be transformed. This class is suggested since *Equals* directly depends on *BuildException* class, which is involved in cycle $C1$. Thus, the optimal class to break cycle $C1$ is suggested, i.e., *Path*. Although CCS(*Equals*) is decreased to 127, a subcycle exists that is formed by 109 classes. The second iteration of the process is triggered and the highest ranked class is *FileUtils*, since it is the optimal class to break the subcycle. After the DIP enforcement on *FileUtils*, the CCS(*Equals*) value is diminished to 5. The same steps of the process are followed in *Contains* and *HasFreeSpace* CCs, since they share the same Level 1 dependencies, and the CCS values for each CC are presented in Table 1.

The process identifies *Path* and *FileUtils* as the best candidates to eliminate cyclic dependencies since they import to the CCs 165–168 unnecessary classes. This occurs due to static calls to both classes. However, the subcycle is identified after the larger cycle was first untangled. In order for the CCs to be compiled without errors, the software engineer identified that *FileUtils* class was statically called

[1] http://students.csd.auth.gr/econst/dl/ReCompId.zip.
[2] A detailed report can be found at http://students.csd.auth.gr/econst/dl/IPL.pdf.

and according to DIP, he refactored the source code and implemented the *FileUtilsInterface* and modified *FileUtils* to depend only on *FileUtilsInterface*. After the aforementioned modification, the CCs are compiled without errors and can be incorporated to another system. Overall, the comprehension of the initial CCs would require excessive effort. Instead, our approach and tool facilitate this task since they reduce the effort required for systems' investigation and transformation in order to diminish CCS values and facilitate their reuse.

OpenProj suffers from CCS explosion, as shown in Fig. 2b, since it contains two cycles with CCS values equal to 402 and 709. The CC(*Duration*) and CC(*Merge*) are examined, where *Duration* stores durations and *Merge* provides functionality about operations on time intervals. Both classes share the same CCS value of 402 and they are provided as input to the process. For *Duration*, the first iteration of the process identifies *SessionFactory* that participates in cycle $C1$ in order to break the cycle. The DIP enforcement on *SessionFactory* results to CCS(*Duration*)=16. However, *Duration* class is responsible for a subcycle of 4 classes and in the second iteration of the process *Duration* is identified as candidate for refactoring. Since the CCS value is manageable and further modifications of the component are unnecessary, the process terminates. The process is repeated for *Merge* class and it is identical to the process application on *Duration* due to common dependencies. As shown in Table 1, the CCS value is decreased to 23 and it is no longer involved in cyclic dependencies. In this case, the identified key class *SessionFactory* can be removed since it involves crosscutting concern functionality about a user session (e.g., username). This functionality is independent of the component intended functionality and can be removed, since reusing *Duration* or *Merge* does not require reusing the session handling mechanism of OpenProj.

5. Threats to validity

Our approach for key classes identification does not consider the type of dependency between classes. A possible threat to internal validity is to recommend key classes with functionality related to the origin class functionality. In future research, we plan to introduce further criteria in the identification of key classes by considering their specificity with respect to the origin class. Another threat to the validity of our approach is when DIP cannot be applied to key classes, e.g., GUI classes. With regard to external validity, we present the results for two mid-sized applications. The proposed methodology for handling the CCS explosion problem requires an exhaustive search of the system's classes. Thus, the performance of our approach decreases for large systems. Finally, we applied our approach only to systems written in Java and we plan to support other object-oriented languages in order to verify the generalization of our results.

6. Conclusion

In this paper we introduce a semi-automated tool-assisted approach to assist source code reuse. We present a method to overcome structural complexity related barriers to extract reusable components. The proposed process guides the software engineer to comprehend the system structure by identifying and transforming key classes that are responsible for large component sizes with ReCompId tool. In future work, we will introduce further criteria to support larger systems and extend our approach beyond component extraction.

References

[1] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Cimpan, H. Verjus, Towards a process-oriented software architecture reconstruction taxonomy, in: Proc. of Eur. Conf. on Softw. Maint. and Reeng., 2007, pp. 137–148.
[2] B. Foote, J. Yoder, Big ball of mud, in: Proc. of Conf. on Patterns Lang. of Prog., 1997, pp. 653–692.
[3] R. Martin, M. Martin, Agile Principles, Patterns, and Practices in C#, Prentice Hall, 2006.
[4] J. Falleri, S. Danier, J. Laval, P. Vismara, S. Ducasse, Efficient retrieval and ranking of undesired package cycles in large software systems, in: Proc. of Int. Conf. on Objects, Models, Compon., Patterns, 2011, pp. 260–275.
[5] H. Melton, E. Tempero, An empirical study of cycles among classes in java, Empir. Softw. Eng. 12 (4) (2007) 389–415.
[6] H. Washizaki, Y. Fukazawa, A technique for automatic component extraction from object-oriented programs by refactoring, Sci. Comput. Program. 56 (1–2) (2005) 99–116.
[7] A. Marx, F. Beck, S. Diehl, Computer-aided extraction of software components, in: Proc. of Working Conf. on Reverse Eng., 2010, pp. 183–192.
[8] X. Wang, X. Yang, J. Sun, Z. Cai, A new approach of component identification based on weighted connectivity strength metrics, Inf. Technol. J. 7 (1) (2008) 56–62.
[9] S. Allier, H.A. Sahraoui, S. Sadou, Identifying components in object-oriented programs using dynamic analysis and clustering, in: Proc. of the Conf. of the Cent. for Adv. Stud. on Collab. Res., 2009, pp. 136–148.
[10] S.D. Kim, S.H. Chang, A systematic method to identify software components, in: Asia-Pac. Softw. Eng. Conf., 2004, pp. 538–545.
[11] J.K. Lee, S.J. Jung, S.D. Kim, W.H. Jang, D.H. Ham, Component identification method with coupling and cohesion, in: Asia-Pac. Softw. Eng. Conf., 2001, pp. 79–86.
[12] R. Tarjan, Depth first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.