(Automated) Software Modularization Using Community Detection

Klaus Marius Hansen(✉) and Konstantinos Manikas

Department of Computer Science (DIKU), University of Copenhagen, Copenhagen, Denmark
{klausmh,kmanikas}@di.ku.dk

Abstract. The modularity of a software system is known to have an effect on, among other, development effort, change impact, and technical debt. Modularizing a specific system and evaluating this modularization is, however, challenging. In this paper, we apply community detection methods to the graph of class dependencies in software systems to find optimal modularizations through communities. We evaluate this approach through a study of 111 Java systems contained in the Qualitas Corpus. We found that using the modularity function of Newman with an Erdős-Rényi null-model and using the community detection algorithm of Reichardt and Bornholdt improved community quality for all systems, that coupling decreased for 99 of the systems, and that coherence increased for 102 of the systems. Furthermore, the modularity function correlates with existing metrics for coupling and coherence.

Keywords: Software architecture · Module structure · Software modularity

1 Introduction

The way a software system is designed and structured influences both the system's development and runtime qualities. In particular, modularity is the quality that encapsulates interdependence within parts (modules) of a system and independence among parts of the system. Good modularization also provides abstraction, information hiding, and specify interfaces [1]. Software modularity provides more benefits than mere logical structuring. It can arguably reduce development effort, minimize impact of change, and reduce technical debt [2,3]. In particular low coupling among modules and high coherence within modules is important.

An optimal (automated) modularization of object-oriented software systems is perceived as a challenge. In this paper, we use community detection methods both for optimising module structure and also for measuring modularity. We apply these methods to a set of open source systems and compare them with existing validated modularity metrics [3,4] . Our findings show that using community detection (in particular Newman modularity with an Erdős-Réyni null model) optimises the modularity of systems and (tentatively) that community detection metrics may be used to measure modularity.

D. Weyns et al. (Eds.): ECSA 2015, LNCS 9278, pp. 95–102, 2015.
DOI: 10.1007/978-3-319-23727-5_8

2 Background and Related Work

Abdeen et al. [4] define a module as "a group of programs and data structures that collaborate to provide one or more expected services to the rest of the software". Moreover they define a set of modularity principles for object-oriented systems and propose metrics for quantifying whether the principles are fulfilled. Li et al. [3] apply the metrics proposed by [4] as a measure of architectural technical debt.

Several authors have investigated automated modularization. Abdeen et al. [5] investigated how to automatically improve modularization of a software system while preserving original design decisions by through a genetic programming approach. Praditwong et al. [6] similarly described modularization as a multi-objective search problem and empirically demonstrated improvements over a single-objective optimisation strategy. Barros et al. [7] used a heuristic search approach to investigate a restructuring of Ant, but found that optimising according to commonly used coupling and coherence metrics led to complex designs.

Finally, to our knowledge none have addressed modularization as a community detection problem. We discuss this and the existing modularity metrics we apply in the following two sections.

2.1 Software Modularization Metrics

While a large number of metrics for object-oriented software deal with coupling and cohesion at the class level, there are few defined metrics that work on the module or package level [4]. Martin [8], Sarkar et al. [9], and Abdeen et al. [4] present metrics that do work on package level.

Abdeen et al.'s metrics fulfill Briand et al.'s properties of coupling and coherence metrics [10]. Furthermore, Li et al. [3] showed that one coupling metric of Abdeen et al. (Index of Package Changing Impact (IPCI)) and one coherence metric (Index of Package Goal Focus (IPGF)) correlated with a measurement of architectural debt in a set of open source project. We thus chose IPCI and IPGF as metrics for coupling and coherence respectively in our study.

In the following, we will briefly review these. Here P is the set of packages of a system, $clients(p)$ for a package $p \in P$ is the set of packages that contain classes with use/extend dependencies on p, $inint(p, q)$ is the set of classes in $p \in P$ that classes in $q \in P$ have use/extends dependencies on, and $inint(p)$ is the set of classes in $p \in P$ that classes in other packages use/extend.

Index of Package Changing Impact (IPCI). IPCI measures coupling as the average proportion of packages that do not change if a package changes. A value close to 0 implies a high degree of coupling among packages while a value close to 1 indicates the opposite. IPCI can be calculated as 1 minus the density of the graph in which each vertex represent a package and edges represent dependencies among packages (induced by use/extends dependencies among classes):

$$IPCI = \begin{cases} \frac{\sum_{p \in P} 1 - \frac{clients(p)|}{|P|-1}}{|P|} = 1 - \frac{\sum_{p \in P} |clients(p)|}{|P|(|P|-1)} & \text{if } |P| > 1, \\ 1 & \text{if } |P| = 1. \end{cases} \quad (1)$$

Index of Package Goal Focus (IPGF). IPGF measures cohesion as the average package focus where the package focus for $p \in P$ is the average proportion of classes in $inint(p)$ that other packages use/extends. A value close to 0 indicates that $q \in clients(p)$ tend to use different sets of classes in p whereas a value close to 1 indicates that $q \in clients(p)$ tend to use the same set of classes in p. Given

$$role(p,q) = \begin{cases} \frac{|inint(p,q)|}{|inint(p)|} & \text{if } |inint(p)| > 0, \\ 1 & \text{if } |inint(p)| = 0. \end{cases}$$

IPGF may be calculated as

$$IPGF = \frac{\sum_{p \in P} \frac{\sum_{q \in clients(p)} role(p,q)}{|clients(p)|}}{|P|} \quad (2)$$

2.2 Community Detection

Many graphs/networks describing real-life phenomena exhibit *community structure* [11]. The structures are partitions of the graph into groups in which there are many edges among vertices in the group, but few edges to vertices outside the group. Community detection methods for finding community structures have been applied in many domains of graph analyses including literary networks, voting patterns, and biology [11,12]. Recently, Gentea and Madsen applied community detection to automated architectural recovery [13], showing improvements over specialized automated software architecture recovery methods.

Community detection algorithms often combine a *quality function* that score a partition and an *optimisation method* that heuristically finds partitions where the quality function is optimal [14]. Our initial idea was to use the Louvain algorithm [14] to find optimal modularizations optimising IPCI and IPGF respectively. However, this is not appropriate since both IPCI and IPGF are degenerate in the sense that they yield a maximum score of 1 for a modularization with all classes in one package (or in one package and with one empty package). We thus focus on more general community detection methods within the Louvain algorithm framework.

Newman's original modularity quality function [15] counts edges within a community and compares this to what would be expected at random:

$$Q = \frac{1}{2m} \sum_{i,j} \left(A_{ij} - P_{ij}\right) \delta(g_i, g_j) \quad (3)$$

Here, m is the number of edges in the graph, A is the incidence matrix for the graph (i.e., $A_{ij} > 0$ if there is an edge between node i and j), P_{ij} is the expected

number of edges between i and j and represents a *null model*, g_i is the community that node i belongs to, and $\delta(g_i, g_j)$ is Kronecker's δ (i.e., $\delta(g_i, g_j) = 1$ if $g_i = g_j$, 0 otherwise). Thus, Q becomes high (close to 1) if communities have many intra-edges compared with the random model.

In our study, we consider an Erdős-Réyni null model in which random graphs $G(n, p)$ with n vertices are created by linking pairs of vertices, i, j, with probability p. This is a simple null model that essentially models that vertices within a community are linked at random. If d is the density of a graph and n is the number of vertices in the graph, then the null model of the graph is $G(n, d)$.

3 Experimental Design

In the following section we explain the design of our study. We conducted a technology-oriented quasi-experiment [16].

3.1 Research Questions

The aim of this study can be summarized by the following research questions:

> *RQ1: How can we modularize an object-oriented software system in an automated way such that this modularity is optimised?*

To address this, we find communities that optimise Newman modularity given an Erdős-Réyni null model (NMER; cf. Section 2.2) and compare this to IPCI and IPGF (cf. Section 2.1). Our second research question builds on this use of community detection methods:

> *RQ2: Can community detection quality functions be useful as software modularity metrics?*

To address this research question, we test to which extent NMER correlates with IPCI and IPGF and to which extent changes in NMER correlates with changes in IPCI and IPGF.

3.2 Data Collection

To study modularity, we chose to study the software systems contained in the *Qualitas Corpus* [17], a collection of curated, open-source Java systems. The reason we chose this is that it is a curated set of open-source systems for which well-defined versions are available for download and because the systems are medium- to large-sized and thus, arguably, modularity is important for them. We studied the latest release 20130901 containing 111 systems The 111 systems have a median NCLOC of 51,860 and standard deviation of 307,473 NCLOCs.

3.3 Analysis Procedures

We first used the Java ASM byte code manipulation and analysis framework [18] to *extract dependencies* in Rigi Standard Format (RSF; [19]) using the binary version of each system in the Qualitas Corpus. Classes recorded were classes defined in the system or depended upon by the system. Dependencies were found in .class files (superclass, implemented interface, accessed attribute types, classes defining invoked method etc.) using a modification of ASM's `org.objectweb.asm.depend.DependencyVisitor`.

From the RSF file, we created a *dependency graph* using Igraph with classes as vertices and dependencies as edges. The package structure, i.e., the original modularization, was used to create an original partitioning of the graph. Since the system architects only have control over the modularity of classes included in the system, we included only those classes in the graph. The classes belonging to the system were determined as being the ones that were defined in the source folder.

We next computed an *optimised modularization* of the system. The optimised modularization was detected using the Louvain framework[1] for Igraph. We used the Reichardt-Bornholdt quality function [20] with an Erdős-Réyni null model. This is a generalization of Newman modularity that includes a resolution parameter. We set the resolution to 1, thus effectively optimising a Newman modularity function with an Erdős-Réyni null model.

For the original and optimised modularization, we calculated the quality of the modularization using a Newman modularity function with an Erdős-Réyni null model (NMER) using the Igraph Louvain framework and the IPCI and IPGF metrics using our own Python implementation.

To answer Research Question 1, we used three Wilcoxon paired signed-rank test to determine if there was a statistically significant difference between the original and optimised NMER, IPCI, and IPGF measures respectively. We used a Wilcoxon test because measurements on software is not usually normally distributed. We used a paired test because the original and optimised modularizations are paired. For the statistically significant differences, we computed an absolute effect size (i.e., difference in means) and a relative effect size using Cohen's d.

To answer Research Question 2, we computed correlation between original NMER and original IPCI, between original NMER and original IPGF, between optimised NMER and optimised IPCI, and between optimised NMER and optimised IPGF. To do this, we used Spearman's ρ. We used Spearman's ρ (instead of, e.g., Pearson's r) again because data is not expected to be normally distributed.

4 Results

Table 1 shows the p value of Wilcoxon, indicating that the population mean ranks of NMER, IPCI, IPGF respectively differ highly significantly. The differences in

[1] https://github.com/vtraag/louvain-igraph

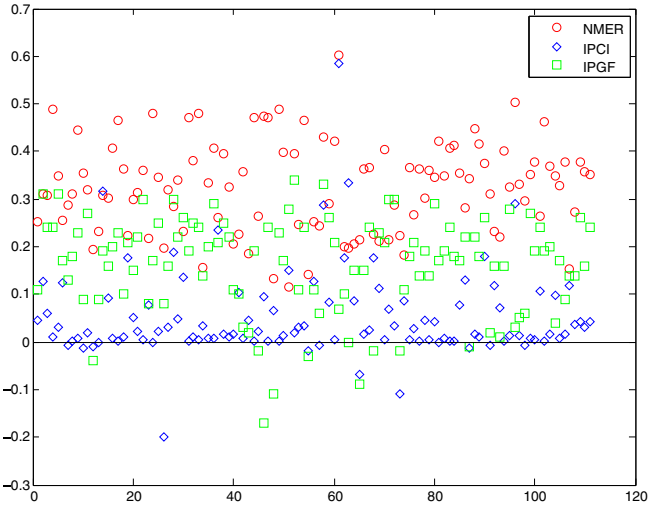

Fig. 1. Plot of the delta of NMER, IPCI, and IPGF after and before optimisation.

mean are 0.3259 for NMER, 0.0529 for IPCI, and 0.1631 for IPGF. Thus, the NMER, IPCI, and IPGF values improve in general even though there are deltas that are negative. Figure 1 shows this as the plot of the delta for NMER (NMER optimised minus NMER original), IPCI, and IPGF. We note that there are some systems with negative delta IPCI (12) and delta IPGF (9). In terms of effect size, calculated as Cohen's d, as can be seen in Table 1, the effect of optimisation is high. In total, we have answered Research Question 1 affirmatively.

To address Research Question 2 we investigate how the IPCI and IPGF metrics correlate with NMER for the original and optimised values. We measure the correlation using Spearman's rank correlation coefficient. The results are shown in Table 2. Except for correlation for original NMER–IPGF, the correlations are highly significant. In terms of the original intent that NMER can be used to measure coupling and coherence, the optimised values are the most relevant and here the correlation is significant.

4.1 Threats to Validity

The software systems that we studied may not be representative samples of the whole population (i.e. all object-oriented open source systems). Moreover, while

Table 1. Significance and effect evaluation for NMER, IPCI, and IPGF.

Score	Wilcoxon p	Mean of original	Mean of optimised	Cohen's d
NMER	5.9863×10^{-20}	0.4301	0.7560	1.6438
IPCI	5.3375×10^{-14}	0.9008	0.9537	0.5150
IPGF	3.2317×10^{-18}	0.7251	0.8882	1.5006

Table 2. Spearman's evaluation for NMER–IPCI, and NMER–IPGF for original and optimised values

Score	Spearman's for original		Spearman's for optimised	
	ρ	p-value	ρ	p-value
IPCI	0.2310	0.0147	0.7116	2.0686×10^{-18}
IPGF	0.1316	0.1685	0.4055	1.0103×10^{-05}

IPCI and IPGF have been validated (for correlation with technical debt in a set of open-source C# software systems [3]), these may not correlate with externally interesting measures for the software systems in the Qualitas Corpus.

5 Conclusion

The modularity and the module structure is an important part of the software architecture of software systems. It has an effect on, among other, development effort, change impact, and technical debt. In this paper we investigate means of optimising software modularity by automated partitioning the classes of object-oriented software systems using community detection methods. Moreover, we investigate the use of quality functions of community detection methods to measure software modularity.

In particular, we propose the use of Newman modularity with an Erdős-Rényi null-model for measuring software modularity and the Reichardt and Bornholdt community detection algorithm that propose a convenient framework for community detection and thus for optimised software modularity. We investigate this in the context of 111 systems contained in the Qualitas Corpus. Our results reveal that our optimisation improved Newman modularity for all systems, that coupling decreased for 99 of the systems, and that coherence increased for 102 of the systems.

References

1. Baldwin, C.Y., Clark, K.B.: Design Rules: The Power of Modularity, vol. 1. MIT Press, Cambridge (2000)
2. Wilkie, F., Kitchenham, B.: Coupling measures and change ripples in C++ application software. Journal of Systems and Software **52**(23), 157–164 (2000)
3. Li, Z., Liang, P., Avgeriou, P., Guelfi, N., Ampatzoglou, A.: An empirical investigation of modularity metrics for indicating architectural technical debt. In: Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures, QoSA 2014, New York, NY, USA, pp. 119–128. ACM (2014)
4. Abdeen, H., Ducasse, S., Sahraoui, H.: Modularization metrics: Assessing package organization in legacy large object-oriented software. In: 2011 18th Working Conference on Reverse Engineering (WCRE), pp. 394–398, October 2011
5. Abdeen, H., Sahraoui, H., Shata, O., Anquetil, N., Ducasse, S.: Towards automatically improving package structure while respecting original design decisions. In: 2013 20th Working Conference on Reverse Engineering (WCRE), pp. 212–221. IEEE (2013)

6. Praditwong, K., Harman, M., Yao, X.: Software module clustering as a multi-objective search problem. IEEE Transactions on Software Engineering **37**(2), 264–282 (2011)
7. de Oliveira Barros, M., de Almeida Farzat, F., Travassos, G.H.: Learning from optimization: A case study with apache ant. Information and Software Technology **57**, 684–704 (2015)
8. Martin, R.C.: The tipping point: Stability and instability in OO design. Dr Dobb's, March 2005
9. Sarkar, S., Kak, A.C., Rama, G.M.: Metrics for measuring the quality of modularization of large-scale object-oriented software. IEEE Transactions on Software Engineering **34**(5), 700–720 (2008)
10. Briand, L.C., Daly, J.W., Wüst, J.: A unified framework for cohesion measurement in object-oriented systems. Empirical Software Engineering **3**(1), 65–117 (1998)
11. Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. Phys. Rev. E **69**(2), 026113 (2004)
12. Mucha, P.J., Richardson, T., Macon, K., Porter, M.A., Onnela, J.P.: Community structure in time-dependent, multiscale, and multiplex networks. Science **328**(5980), 876–878 (2010)
13. Gentea, A., Madsen, T.: Using community detection methods for automated software architecture recovery. Master's thesis, Department of Computer Science, University of Copenhagen, September 2014
14. Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment **2008**(10), P10008 (2008)
15. Newman, M.E.: Finding community structure in networks using the eigenvectors of matrices. Physical Review E **74**(3), 036104 (2006)
16. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer (2012)
17. Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J.: Qualitas corpus: A curated collection of java code for empirical studies. In: 2010 Asia Pacific Software Engineering Conference (APSEC 2010), pp. 336–345, December 2010
18. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A code manipulation tool to implement adaptable systems. In: Adaptable and Extensible Component Systems, Grenoble, France, November 2002
19. Wong, K.: Rigi Users Manual. Department of Computer Science, University of Victoria, July 1996. http://www.rigi.cs.uvic.ca/downloads/rigi/doc/user.html
20. Reichardt, J., Bornholdt, S.: Statistical mechanics of community detection. Physical Review E **74**(1), 016110 (2006)