Refactoring of Object-oriented Package Structure Based on Complex Network

Youfei Huang, Yuhang Chen, Zhengting Tang, Liangyu Chen, Ningkang Jiang*,
Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University,
Shanghai, China,
nkjiang@sei.ecnu.edu.cn

***Abstract*—A software system is usually developed with multiple modules. However, its structure is continuously modified during software evolution, resulting in poor maintainability and understandability. Therefore, software evolution must accompany system refactoring. This paper describes an optimization approach for package structure according to complex network theory. First, we analyze the relations between classes and build the class dependency graph. Second, we propose a community detection algorithm to recombine the classes and optimize system cohesion and coupling without changing the external functionality. Third, by comparing the original and optimized package structure, the two dimensions of splitting the package and moving classes between packages identify package refactoring opportunities. In addition, we evaluate the impact of the above approach on package quality in terms of package reusability and instability. We design experiments on 10 open-source Java software projects to verify the effectiveness of our approach.**

***Index Terms*—Refactoring;complex network;community detection;cohesion;coupling**

I. Introduction

During the software life cycle, new requirements will emerge inevitably, and software modifications to implement new requirements may not conform to object-oriented programming specifications [1]. In addition, the tight development cycle may lead to poor design decisions [2]. The differences between the system and the original design increase in the long run, and the quality of the system becomes increasingly poor. It becomes more and more challenging to maintain the existing software. In order to avoid the fate of software system corruption, fragmentation and even disintegration, choosing a proper refactoring operation is a feasible method. Refactoring can adjust the software architecture without changing the code's external behavior, optimize the existing code, and extend the software system's life [3].

In the process of software development and maintenance, the problem of Too Large Packages (TLP) or Too Small Packages (TSP) may occur in the system. When a package contains more than 30 classes or exceeds $27,000$ code lines [4], it can be considered as TLP. The number of classes or lines of code in a package that the TSP should contain is not explicitly given in [4]. According to previous research, when the number of classes in a package is 4 or less [5, 6], it is usually not worth the effort to maintain them.

DOI reference number: 10.18293/SEKE2022-164

Complex network theory has been applied to many fields, such as social networks [7, 8], computer networks [9], and biological networks [10]. One of the notable features of complex networks is their community structure [11, 12]. Software network built with classes as nodes and dependencies between classes as edges is the product of characterizing software systems as complex networks, so it has community structure property [13]. Based on complex network theory, we propose a system-level automatic package refactoring approach that abandons the original package structure of the system and re-modularizes the system. First, we represent the software system as a class dependency graph. Second, based on the definition of TLP and TSP, we propose a community detection algorithm to detect the community structure corresponding to the optimized package structure. Third, by comparing the optimized package structure with the original package structure, we give refactoring suggestions. Through tests on 10 open source software systems, it is verified that our refactoring approach is meaningful and positively impacts the cohesion of the system while reduces the coupling of the system. The primary contributions of this study are summarized as follows.

- From the perspective of optimizing system structure, a controllable community detection algorithm that divides the size of sub-communities is proposed, which can obtain the best class distribution from the system-level, and eliminate too large and small packages in the system.
- Several open-source software systems are used to evaluate our approach. The evaluation metrics based on cohesion and coupling metrics are compared with previous studies, and we also assess the changes in system reusability and stability after refactoring.

The remainder of the paper is organized as follows. Section II summarizes the related work. Section III presents the package refactoring algorithm. In Section IV, we design experiments to verify the effectiveness of our approach. Section V is the conclusion.

II. Related Work

A. Package-level Refactoring

Over the last three decades, software engineers have proposed several semi-automatic and full-automatic refactoring methods to improve software quality. Depending on the kind of entities selected in refactoring, there are three main types of refactoring at different granularities: package level, class level, method and property level. In this paper, we focus on package level refactoring. Pan et al. [14] represented the package, class and their dependencies with weighted bipartite software networks and proposed a guidance community detection algorithm to optimize the package structure of the software system. Mi et al. [15] divided the dependency relationships between classes into five types to build a class dependency graph, and then proposed a cohesion metric at the package level, according to this metric to move class between packages. Zhou et al. [16] applied Mi's approach to build software networks, proposed a coupling metric of packages and improved the structure of packages considering changes in cohesion and coupling values. Bavota et al. [5] combined semantic and structural metrics to generate a class-by-class matrix, where the values in the matrix indicate the likelihood of two classes being in the same package, after which the strongly related class chains in the matrix are extracted, and the classes in one class chain are placed in the same package. However, their approach only focuses on refactoring one package at a time, and incrementally re-modularizing a software system. Abdeen et al. [17] and Chhabra et al. [18] used the multi-objective Non-Dominated Sorting Genetic Algorithm to optimize system structure by moving classes between packages, while respecting the original package organization as much as possible, to increase cohesion and reduce coupling and cyclic connectivity.

B. Evaluation Metrics of Package Quality

Since the software is frequently changing, software designers should assess software quality periodically. The evaluation standard is the authoritative software metrics proposed by research and engineering. It plays an important role in many fields in the life-cycle of the software, including predicting software defects and maintenance costs [19], and if they are appropriately selected and applied, improvements can be identified and quantified. Wang et al. [2] used the Quality Model for Object-Oriented Design (QMOOD) metric proposed by Harrison et al. [20] to evaluate the improvement in reusability, flexibility, and understandability of the system after their refactoring. QMOOD is a quality metric at the class level in object-oriented design, and Singh et al. [21] proposed the Quality Metric of Package Level in Object-Oriented Design (QMPOOD) with quality attributes including reusability, flexibility, functionality, understandability, extendibility, and effectiveness, and later they gave a specific method to calculate the reusability of software system packages in [22]. Chong et al. [23] presented a weighted complex network to represent the structural characteristics of object-oriented software systems and used 40 object-oriented software systems for experiments to evaluate the maintainability and reliability of software systems. Martin et al. [24] proposed software package metrics based on object-oriented design principles, including eight metrics: efferent coupling (C_e), afferent coupling (C_a), instability (I), number of abstract classes (Na), number of classes (N_c), abstractness (A), the distance from the main sequence (D) and the normalized distance from the main sequence (D_n).

III. METHODOLOGY

A. Problem Formulation

We use the code snippet shown in Fig.1 as a motivation example to formulate our problem. In Fig.1, there are 11 classes, which are divided into 4 packages. Classes do not exist independently, but some classes are more dependent on classes in other packages. Therefore, there are "bad smells" in the code.

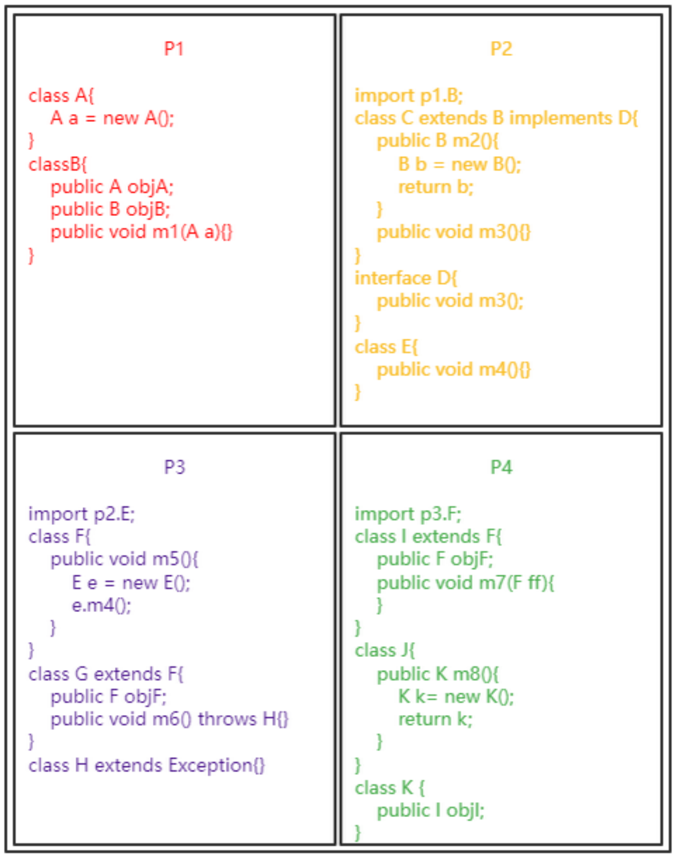

P1
```
class A{
    A a = new A();
}
classB{
    public A objA;
    public B objB;
    public void m1(A a){}
}
```
P2
```
import p1.B;
class C extends B implements D{
    public B m2(){
        B b = new B();
        return b;
    }
    public void m3(){}
}
interface D{
    public void m3();
}
class E{
    public void m4(){}
}
```
P3
```
import p2.E;
class F{
    public void m5(){
        E e = new E();
        e.m4();
    }
}
class G extends F{
    public F objF;
    public void m6() throws H{}
}
class H extends Exception{}
```
P4
```
import p3.F;
class I extends F{
    public F objF;
    public void m7(F ff){
    }
}
class J{
    public K m8(){
        K k= new K();
        return k;
    }
}
class K {
    public I objI;
}
```

Fig. 1. Code snippet used as an example to understand the proposed algorithm.

Based on five class dependencies outlined in [15], which are inheritance and implementation, aggregation, parameter, signature, and invocation. We extract the associations between classes in Fig.1 and then build the Class Dependency Graph (CDG), shown in Fig.2.

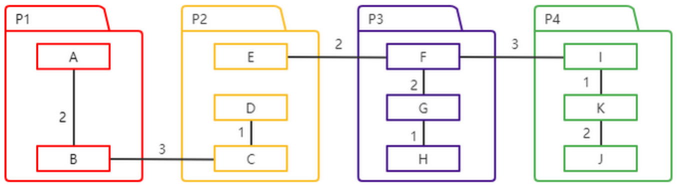
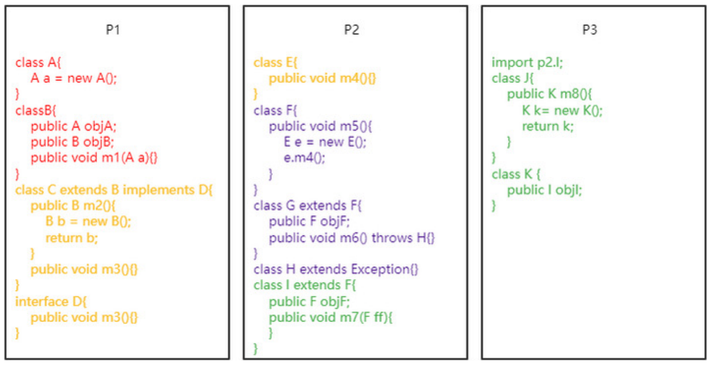


Fig. 2. Class dependency graph



P1

```
class A{
   A a = new A();
}
classB{
   public A objA;
   public B objB;
   public void m1(A a){}
}
class C extends B implements D{
   public B m2(){
      B b = new B();
      return b;
   }
   public void m3(){}
}
interface D{
   public void m3(){}
}
```

P2

```
class E{
   public void m4(){}
}
class F{
   public void m5(){
      E e = new E();
      e.m4();
   }
}
class G extends F{
   public F objF;
   public void m6() throws H{}
}
class H extends Exception{}
class I extends F{
   public F objF;
   public void m7(F ff){
   }
}
```

P3

```
import p2.I;
class J{
   public K m8(){
      K k= new K();
      return k;
   }
}
class K {
   public I objI;
}
```

Fig. 3. Optimized package structure.

After refactoring, we get three new packages, shown in Fig.3. The optimized package structure gains a better modularity Q, increased from 0.2699 to 0.5450.

For our refactoring approach, this paper mainly studies the following research questions:

- RQ_1: Can our approach alleviate the design problems and get meaningful refactoring?
- RQ_2: Does our approach have more advantages compared with other refactoring approaches?
- RQ_3: Besides cohesion and coupling, does our approach improve other package design metrics?

B. Method Overview

Our refactoring approach is shown in Fig.4. First, we model the package topology of an object-oriented software system as a weighted software network, with classes as vertices and dependencies between classes as the edges of the network. Second, based on complex network theory, this paper uses a community detection algorithm to recombine the classes and find the communities corresponding to the optimized packages. Third, by comparing the optimized package structure with the original package structure, we obtain the package refactoring opportunities.

C. Package Refactoring Algorithm

With Java software systems as research objects, we analyze the bytecode files of the software system and regard classes and their dependencies as entities. In order to improve the package structure, the refactoring approach in this paper regards each class in the system as an independent community, and the classes in the software system are gradually reaggregated to form several packages by using the community detection algorithm. The number of sub-communities of the community detection algorithm, that is, the total number of packages after system optimization, is not specified in advance, so the number of packages may be different from the original system.

Community detection, also known as community discovery, is a technique used to reveal network aggregation behavior. The nodes in the same community are densely connected, and the connections between nodes in different communities are sparse. Newman et al. [25] proposed to calculate the modularity of unweighted undirected networks. Modularity is a property of a network and a measure of the quality of a particular division of a network. However, this paper builds a weighted software network. Therefore, we appropriately modify the calculation method of modularity and propose a weighted modularity Q, which is defined as

$$\begin{aligned} Q &= \frac{1}{2W}\sum_{ij}\left(w_{ij}-\frac{h_i h_j}{2W}\right)\delta\left(c_i,c_j\right) \\ &= \sum_{p=1}^{n}\left[\frac{W_p}{W}-\left(\frac{H_p}{2W}\right)^2\right], \end{aligned} \tag{1}$$

where W is the sum of all the edge weights, w_{ij} is the weight of the edge between node i and node j, h_i is the sum of the weights of all edges connected to node i, c_i and c_j are the communities to which nodes i and j belong, respectively. If i and j are in the same community, then $\delta\left(c_i,c_j\right)=1$, otherwise $\delta\left(c_i,c_j\right)=0$. And n is the number of communities in the network, W_p is the sum of the weights of the edges within community c_p, H_p is the sum of the weights of all edges connected to community c_p. Based on (1), when merging communities, the modularity change of the system can be calculated by (2), $H_{in}(c_i,c_j)$ represents the sum of all the edge weights in the sub-community formed by the merging of community c_i and c_j.

$$\Delta Q_{ij}=\begin{cases}\frac{H_{in}(c_i,c_j)-H_iH_j}{2W} & \text{if } c_i \text{ connects with } c_j \\ 0 & \text{otherwise,}\end{cases} \tag{2}$$

We propose a community detection algorithm, which takes into account the goal of this package refactoring and avoids too large or small packages. In order to prevent excessive software refactoring, when a community and multiple target communities have the same modularity change after merging, we prioritize merging the community pair with entities that are all defined in the same original package to maintain the original design as much as possible. The refactoring algorithm is shown in Algorithm 1:

Algorithm 1 Package refactoring algorithm

```
Input: The adjacency matrix $M_{n\times n}$ of $CDG$.
Output: Community set $C$; Weighted modularity $Q$.
  1. Let the weighted modularity $Q=0$. The number of initial
  communities is the number of classes in the system.
  2. We calculate the change of the $Q$ of the system after the
  sub-community is merged according to (2), and calculate
  the modularity increment matrix $\Delta Q$.
  3. we find the maximum element $\Delta Q_{ij}$ in $\Delta Q$, and merge
  the two communities $C_i$ and $C_j$. And then, we recalculate
  the modularity increment matrix according to Step 2.
  Communities are gradually aggregated until the maximum
  element $\Delta Q_{ij}\leq 0$. Then $Q$ reaches the optimum. We obtain
  the community set $C_1$, and the weighted modularity value
  $Q=Q_1$.
  4. According to the definition of TLP, we split the large
  communities existing in the $C_1$ as follows: for the large
```

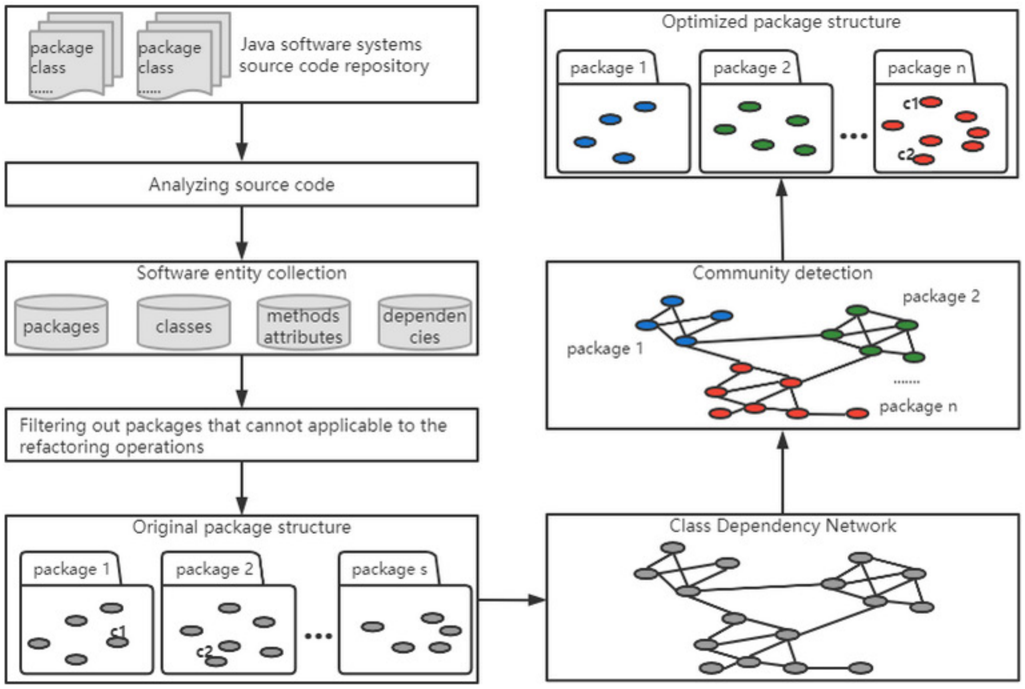

Java software systems source code repository
package
class
Analyzing source code
Software entity collection
packages
classes
methods attributes
dependencies
Filtering out packages that cannot applicable to the refactoring operations
Original package structure
package 1
package 2
package s
c1
c2
Class Dependency Network
Community detection
package 1
package 2
package n
Optimized package structure
package n

Fig. 4. The workflow of the refactoring approach.

community, we delete the edges with smaller weight in turn. In a splitting round, the edge with the smallest weight is chosen, and the community is split into several connected components, where each component has no more than 30 nodes. If not, split recursively. Finally, we obtain the community set C_2, and the weighted modularity value $Q = Q_2$.
5. For TSP in C_2, we relocate them to other communities with the following merging method: according to the reduction value of modularity value Q, we merge the small sub-community with the target community with the smallest reduction of Q, at the same time, the number of nodes in the newly generated community cannot exceed 30, otherwise we merge with the target community with the second smallest reduction of Q.

IV. Experiments

A. Data Sets

The experiments are conducted on a computer with i5-3230M CPU, 8G DDR3 Memory, Windows 10. We select 10 open-source Java software systems to verify the effectiveness of the refactoring approach. Our choice of software systems is not random, as they are projects in different application fields. In the future, we will use more systems to verify the effectiveness of our approach. To remove the unrelated files, we filter the experimental data from the following three aspects:

- Only the classes in top-level packages are considered for experiments.
- Utility modules do not participate in the refactoring process.
- Third-party libraries are excluded since they are not parts of software systems.

TABLE I
Detailed information about 10 Java software systems

System	System	PN	CN	EN
$S0$	Cglib-nodep 3.2.6	9	198	961
$S1$	Codec 1.15	7	139	278
$S2$	Emma 2.0.5312	10	140	506
$S3$	Empire-db 2.5.0	21	178	1097
$S4$	GistoolkitSource 2.8.1	64	504	2228
$S5$	ITtracker 3.1.5	38	422	1803
$S6$	Rng 1.3	19	346	729
$S7$	Roller 5.1.1	61	541	2707
$S8$	Tomcat 9.0.1	42	619	1589
$S9$	XMLgraphics-commons-2.6	35	363	801

Table I summarizes the information of 10 systems after pre-processing, including name and version number, number of package(PN), number of class(CN), number of edges in the software network (EN).

B. Changes of the package structure

We follow the steps in Fig.4 to perform the refactoring operation. Table II shows the change of package structure before and after refactoring. Note that PN represents the number of packages, TLP and TSP represent the threshold of too large and small package, respectively, It is observed that our approach can solve the problem of too small packages and too large packages in the software system.

C. Evaluations of Cohesion and Coupling Metrics

In our experiments, we use *COHM* metric [15] to measure the cohesion of packages and *COUM* metric [16] to evaluate

TABLE II
DETAILED INFORMATION ABOUT PACKAGE STRUCTURE

System	Before refactoring			After refactoring		
	PN	TLP	TSP	PN	TLP	TSP
$S0$	9	1	2	8	0	0
$S1$	7	1	1	7	0	0
$S2$	10	1	4	5	0	0
$S3$	21	1	9	13	0	0
$S4$	64	2	27	36	0	0
$S5$	38	2	10	25	0	0
$S6$	19	5	5	19	0	0
$S7$	61	3	20	32	0	0
$S8$	42	6	9	35	0	0
$S9$	35	1	8	28	0	0

the coupling between packages. A higher value of *COHM* indicates a better cohesion of the package. And the lower value of *COUM* indicates the better coupling of the package. We apply these two metrics to the software systems to measure the improvement by refactoring. Table III records the detailed change of *COHM* and *COUM*. We can see the cohesion are increased and the couping are decreased on all systems, which means the structures of all systems are optimized and the refactorings are useful. Thus, we answer the question RQ_1.

TABLE III
CHANGES IN COHESION AND COUPLING METRIC VALUES

System	COHM	COUM
	Before/After/Diff.	Before/After/Diff.
$S0$	0.273/0.365/+0.092	0.163/0.109/-0.054
$S1$	0.409/0.636/+0.227	0.192/0.072/-0.120
$S2$	0.360/0.516/+0.156	0.236/0.161/-0.075
$S3$	0.130/0.226/+0.096	0.404/0.370/-0.034
$S4$	0.148/0.449/+0.301	0.516/0.222/-0.294
$S5$	0.099/0.284/+0.185	0.685/0.445/-0.240
$S6$	0.328/0.504/+0.176	0.412/0.203/-0.209
$S7$	0.174/0.301/+0.127	0.495/0.227/-0.268
$S8$	0.415/0.579/+0.164	0.273/0.191/-0.082
$S9$	0.372/0.634/+0.262	0.294/0.114/-0.180

D. *Comparison with Previous Research*

We have re-implemented Zhou's [16] work and Abdeen's [17] work on package refactoring. Zhou's approach optimizes the package structure by considering both cohesion and coupling measures, moving the class between packages, and finally selecting the package with better cohesion and coupling as the target package for refactoring the current class. Abdeen's approach is a more conservative optimization of the package structure, using the Non-Dominated Sorting Genetic Algorithm to refactor the package. We compare our refactoring approach with theirs, and Table IV presents the specific changes of cohesion and coupling after refactoring with using three different approaches. The optimal values are presented in bold. It is clear that the cohesion and coupling obtained by our method outperform consistently the other two methods. Therefore, we answer the question RQ_2.

TABLE IV
COMPARISON OF THREE REFACTORING APPROACHES

System	COHMaf	COUMaf
	Our/Zhou/Abdeen	Our/Zhou/Abdeen
$S0$	**0.365**/0.331/0.333	**0.109**/0.159/0.135
$S1$	**0.636**/0.562/0.543	**0.072**/0.080/0.089
$S2$	**0.516**/0.406/0.439	**0.161**/0.174/0.176
$S3$	**0.226**/0.181/0.204	**0.370**/0.371/**0.370**
$S4$	**0.449**/0.249/0.289	**0.222**/0.276/0.337
$S5$	**0.284**/0.140/0.196	**0.445**/0.492/0.523
$S6$	**0.504**/0.372/0.364	**0.203**/0.301/0.305
$S7$	**0.301**/0.223/0.245	**0.227**/0.344/0.366
$S8$	**0.579**/0.449/0.493	**0.191**/0.222/0.229
$S9$	**0.634**/0.437/0.417	**0.114**/0.216/0.199

E. *Evaluations of Reusability and Instability Metrics*

In this section, we investigate whether our refactoring approach optimizes the design quality of the package, besides cohesion and coupling metric. We focus on reusability and instability. Reusability [22] reflects the ability of a design to be reused in multiple contexts. The higher its value, the higher the reusability of the package. Martin proposed instability in [24] to describe the system stability, the lower its value, the more stable of the package.

Table V records the changes in reusability and instability after refactoring. One can observe that the reusability values are increased while the instability values are decreased. This means the design of all software systems is improved after refactoring. We also visualize the change of reusability and instability in Fig.5 and Fig.6, respectively, for a better understanding the change. Therefore, our approach not only improves the cohesion and coupling metrics, but also improves the design quality of system packages in terms of reusability and instability. So we answer the question RQ_3.

TABLE V
CHANGES IN REUSABILITY AND INSTABILITY METRIC VALUES

System	Reusability	Instability
	Before/After/Diff	Before/After/Diff
$S0$	16.071/18.115/+2.044	0.710/0.549/-0.161
$S1$	17.182/17.290/+0.108	0.671/0.557/-0.114
$S2$	9.490/12.189/+2.699	0.519/0.489/-0.030
$S3$	7.312/14.230/+6.918	0.625/0.576/-0.049
$S4$	6.952/17.468/+10.516	0.515/0.510/-0.005
$S5$	9.920/15.941/+6.021	0.693/0.497/-0.196
$S6$	13.604/13.733/+0.129	0.676/0.594/-0.082
$S7$	8.072/15.601/+7.529	0.599/0.533/-0.066
$S8$	11.444/13.826/+2.382	0.530/0.484/-0.046
$S9$	8.927/11.269/+2.342	0.515/0.386/-0.129

F. *Threats to Validity*

The internal validity threat to our study is that the weights of five dependencies between classes are equal in building class dependency network, whereas the priorities of the five dependencies should be different according to other classical theories of software. But to the best of our knowledge, no research has given the specific weight assignments that these five dependencies apply to various systems or even a

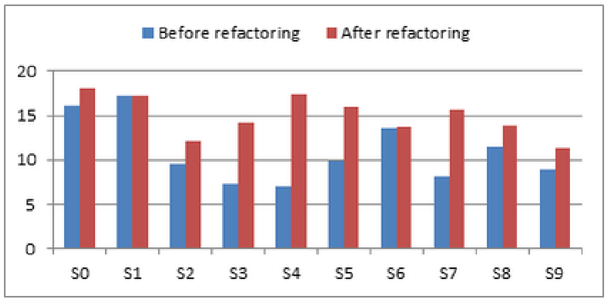


Fig. 5. Reusability change on 10 systems after refactoring.

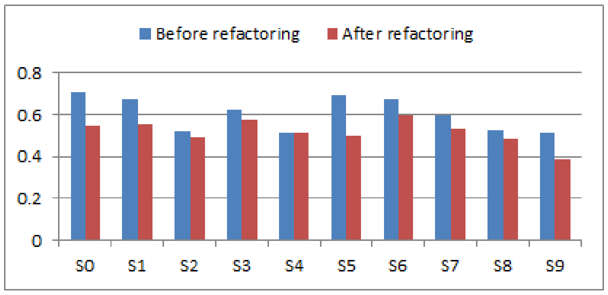


Fig. 6. Instability change on 10 systems after refactoring.

rough range. Considering the importance of five dependencies between classes as the same to build a software weighted network, the effectiveness has been proved in [14], so this part of the threat can be mitigated to a certain extent.

The external validity threat to our study is the limitation of software systems chosen in the experiments. In this paper, we mainly use Java software systems, but there are object-oriented software systems developed in other programming languages such as C++, Python, etc. Therefore, applying our research to projects developed in other programming languages may lack the ability to give accurate refactoring recommendations.

V. Conclusion

In this study, we propose a refactoring approach for package structure based on complex network theory. It uses a community detection algorithm to find opportunities for package refactoring, which achieves the optimal class distribution and get no too large or small packages. The paper analyzes five kinds of dependencies between object-oriented software system classes, which are used to build class dependency network, and then perform refactoring operations according to the software design principle of "high cohesion and low coupling". Experimental results demonstrate that our approach can solve the problem of system cohesion and coupling while maintain the external functionality, and improve software stability and reusability.

References

[1] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[2] Y. Wang, H. Yu, Z. Zhu, W. Zhang, and Y. Zhao, "Automatic software refactoring via weighted clustering in method-level networks," *IEEE Transactions on Software Engineering*, vol. 44, no. 3, pp. 202–236, 2017.

[3] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[4] M. Lippert and S. Roock, *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons, 2006.

[5] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, no. 5, pp. 901–932, 2013.

[6] R. A. Bittencourt and D. D. S. Guerrero, "Comparison of graph clustering algorithms for recovering software architecture module views," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 251–254.

[7] J. M. Hofman, A. Sharma, and D. J. Watts, "Prediction and explanation in social systems," *Science*, vol. 355, no. 6324, pp. 486–488, 2017.

[8] H. Ebel, L.-I. Mielsch, and S. Bornholdt, "Scale-free topology of e-mail networks," *Physical review E*, vol. 66, no. 3, p. 035103, 2002.

[9] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[10] S. N. Dorogovtsev, S. N. Dorogovtsev, and J. F. Mendes, *Evolution of networks: From biological nets to the Internet and WWW*. Oxford university press, 2003.

[11] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the national academy of sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[12] M. E. Newman, "Modularity and community structure in networks," *Proceedings of the national academy of sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[13] L. M. Hakik and R. El Harti, "Measuring coupling and cohesion to evaluate the quality of a remodularized software architecture result of an approach based on formal concept analysis," *International Journal of Computer Science and Network Security*, vol. 14, no. 1, pp. 11–16, 2014.

[14] W. Pan, B. Li, B. Jiang, and K. Liu, "Recode: software package refactoring via community detection in bipartite software networks," *Advances in Complex Systems*, vol. 17, no. 07n08, p. 1450006, 2014.

[15] Y. Mi, Y. Zhou, and L. Chen, "A new metric for package cohesion measurement based on complex network," in *2019 8th International Conference on Complex Networks and Their Applications*. Springer, 2019, pp. 238–249.

[16] Y. Zhou, Y. Mi, Y. Zhu, and L. Chen, "Measurement and refactoring for package structure based on complex network," *Applied Network Science*, vol. 5, no. 1, pp. 1–20, 2020.

[17] H. Abdeen, H. Sahraoui, O. Shata, N. Anquetil, and S. Ducasse, "Towards automatically improving package structure while respecting original design decisions," in *2013 20th Working Conference on Reverse Engineering*. IEEE, 2013, pp. 212–221.

[18] Amarjeet and J. K. Chhabra, "Improving package structure of object-oriented software using multi-objective optimization and weighted class connections," *Journal of King Saud University-Computer and Information Sciences*, vol. 29, no. 3, pp. 349–364, 2017.

[19] M. A. A. Mamun, C. Berger, and J. Hansson, "Effects of measurements on correlations of software code metrics," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2764–2818, 2019.

[20] R. Harrison, S. J. Counsell, and R. V. Nithi, "An evaluation of the mood set of object-oriented software metrics," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 491–496, 1998.

[21] V. Singh and V. Bhattacherjee, "Evaluation and application of package level metrics in assessing software quality," *International Journal of Computer Applications*, vol. 58, no. 21, pp. 38–46, 2012.

[22] V. Singh and V. Bhattacherjee, "Assessing package reusability in object-oriented design," *International Journal of Software Engineering and Its Applications*, vol. 8, no. 4, pp. 75–84, 2014.

[23] C. Y. Chong and S. P. Lee, "Analyzing maintainability and reliability of object-oriented software using weighted complex network," *Journal of Systems and Software*, vol. 110, no. 1, pp. 28–53, 2015.

[24] R. C. Martin, J. Newkirk, and R. S. Koss, *Agile software development: principles, patterns, and practices*. Upper Saddle River, NJ: Prentice Hall, 2003.

[25] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Physical review E*, vol. 70, no. 6, p. 066111, 2004.