Abstract  The quality of a software system is largely determined by its internal structures which always degrade over the software evolution. Therefore, the structures have to be reconditioned from time to time. However, the existing methods are very complex and resource-consuming when doing this task. In this paper, we present an approach to recondition the class structures of object-oriented (OO) software systems. It uses attribute-method networks and method-method networks to represent attributes, methods and dependencies between them; It proposes a guided community detection algorithm to obtain the optimized community structures in the method-method networks, which also correspond to the optimized class structures; It also provides a list of refactorings by comparing the optimized class structures with the real class structure in software systems and inspecting the attribute-method networks. The approach is evaluated using the open-source case study, JHotDraw 5.1, and the advantages of our approach are illustrated in comparison with existing methods.

Keywords  object-oriented (OO) software, software refactoring, class refactoring, complex networks, software dependency networks, community detection

1 Introduction

The quality and longevity of a software system is largely determined by its internal structures. However, a system’s original design is rarely prepared for every new requirement which appears over the software system’s lifecycle. Therefore, the software has an intrinsic property to evolve. Because of many constraints such as time, resources, etc. in real software development practices, change has to be made quickly by different people without properly adjusting the system’s global structure [1]. The system becomes more complex and drifts away from its original design, thereby badly lowering the quality of the software [2]. This phenomenon is known as software decay or design drift [3]. Because of this, it is often costly to maintain [4, 5].

Therefore software systems should be reconditioned from time to time throughout their lifecycle. Refactoring as proposed by Fowler in Ref. [6] is viewed as a way that can be a tremendous aid to improve software design and maintainability, both at the time of the original development, as well as during maintenance of legacy code. However, software engineers have to identify spots having a negative impact on the system’s maintainability, and apply appropriate refactorings to remove the so-called “bad-smells” [1, 7]. But deciding which refactoring to apply and where to apply it is a tough question, since refactoring can have undesired side effects on the system’s structures [1].

To assist software engineers to do software refactoring, in this paper, we propose a novel methodology, with reference to the community detection technology in complex networks, to recondition the class structures of object-oriented software systems. First, object-oriented (OO) software systems are represented by two kinds of dependency networks, attribute-method networks and method-method networks. Second, a guided community detection algorithm is proposed to optimize the community structures in
method-method networks, which correspond to the optimized class structures. Third, a list of refactorings will be suggested by comparing the optimized class structures with the corresponding real class structures in software systems and inspecting the attribute-method networks.

The rest of this paper is organized as follows. Section 2 gives the motivation of our study. Section 3 contains a brief summary of the related work. Section 4 describes our approach in detail, with focus on the formal definitions of related software dependency networks, the procedures to refactoring the class structures, and the algorithm we used to optimize the community structures in method-method networks. In Section 5 we present the results of an experiment conducted on a well-known open source case study. We conclude in Section 6.

## 2 Motivation

When analyzing the inter-method dependencies in some OO software systems by method-method networks (which will be defined in section 4.4), we made the following observation: methods belonging to the same classes tend to be cohesive, while methods from different classes tend to be sparser. That is, in the method-method networks there exist some cliques within which the network connections are dense, but between which they are sparser. This is very similar to the community structures [8, 9] that have been studied widely in complex networks. This phenomenon accords with one of basic software principles, high cohesion and low coupling. But at the same time, we also observed that there exist some methods which do not respect such a rule, i.e., they have more dependencies to methods defined in other classes than in the class where they are defined. These methods increase the coupling between classes, thereby degrading the software quality. Therefore if we can detect these methods and place them in better places to increase the cohesion and decrease the coupling, the quality of software system will be improved to a certain degree. Obviously it is a problem belonging to the research content of software refactoring. The class structures are so similar to the community structures in complex networks that it enlightens us to use community detection technology in complex networks to recondition the class structures.

## 3 Related works

In this section we discuss related works: they falls into three categories: software refactoring, community detection technology in complex networks, and research work on software dependency networks.

### 3.1 Software refactoring

There have been various approaches in the field of software refactoring. Byung-Kyoo Kang and James M. Bierman introduced a quantitative framework for software restructuring [10]. In their framework, restructuring decisions are guided by visualization design information and objective criteria. Olaf Seng, Johannes Stammel and David Burkhardt proposed a search-based approach for refactoring class structures of software systems [1]. They used an evolutionary algorithm to identify refactorings that improve class structures. Igor Ivkovic and Kostas Kontogiannis proposed a framework for refactoring software structures using model transformations and quality improvement semantic annotations [11]. Dudzikan and Wlodka presented an integrated approach for restructuring programs written in Java starting from a catalog of bad smells and propose potential solutions to the users [12]. István Gergely Czibula and Gabriela Serban use clustering to improve system’s design [13].

### 3.2 Community detection technology

Complex networks as a branch of complexity science have been recently studied across many fields of science. By simplifying its research object into networks composed of a set of nodes or vertices connected together by links or edges, researchers have found that many real world networks seem to share some features in common, that is small world [14], scale-free [15], etc. One network feature that has been emphasized in recent work is community structure, the gathering of vertices into groups such that there is a higher density of edges within groups than between them [9]. The problem of community detection has been well studied and a lot of algorithms have been proposed such as Kernighan-Lin algorithm [16], spectral portioning [17], hierarchical clustering [18], divisive algorithms [19, 20] and many faster algorithms [8, 21, 22].

### 3.3 Software dependency networks

In recent years, researchers in the field of statistical physics and complex systems used complex software networks (software dependency networks) to represent software systems by taking software components such as methods, classes, packages, etc. as nodes and their dependencies as edges. It provides us a new way to study complex software systems. Also, the research content is mainly involved in discovering the shared topological properties of software dependency networks, the evolution mechanisms of software dependency networks and the metrics for evaluating complexity of software dependency networks. Literature [23] gives a detailed review of the research work in such a new field.
4 The approach

While there are vast amounts of work on software refactoring, many of them are complex and time-consuming. Though this is not the first work on software refactoring, we will cover a different angle: to identify the factorings and refactoring class structures, we represent software systems by attribute-method networks and method-method networks and present a novel guided community detection algorithm. A list of refactorings for class structure refactoring can be obtained by community detection in method-method networks and inspecting the attribute-method networks. Fig. 1 gives a short overview of the workflow of the proposed approach. In the following sections we will detail it.

![Fig. 1 The workflow of the proposed approach](image)

4.1 Source code

In this paper we mainly focus on the OO domains, and take the open-source object-oriented (OSOO) software systems as our research subjects. The rationale is threefold:

(1) a lot of existing research work focus on OO software systems. It is easy to compare the results of the similar research work under the same OO paradigm.

(2) OO has became the most widely used development paradigm since the 1990s. Also, there are a lot of OSOO software systems on the web which can be easily obtained for our research objectives.

(3) OSOO software systems are developed under the OO paradigm. They have a relatively clear internal structure and the elements such as attributes, methods, classes, packages, and their dependencies are amenable to extraction and analysis.

4.2 Software elements collection

Software elements collection is the first step of the proposed approach. It refers to the process to extract software elements such as attributes, methods and their dependencies. We have developed a tool that can be used to analyze compiled Java codes (.class), Java source files (.java), and C++ source files (.cpp). Only two kinds of dependencies are taken into considerations in our approach. They are attribute-method dependency and method-method dependency. The specific definitions of these two kinds of dependencies will be detailed in Section 4.4. Details on how to extract these dependencies is provided in the experiment.

4.3 Data pre-processing

It is of vital importance to distinguish the methods that we have extracted in Section 4.2 according to the role they play in the system’s design before refactoring. Because some methods belonging to design patterns always have a special function and deliberately violate the existing design guidelines such as high cohesive ness and low coupling, they cannot be treated equally as other ordinary methods and will be viewed as special methods. So in our approach, we only focus on methods that respect general guidelines. So the methods belonging to design patterns and its like will be detected first and marked as special methods. It means that they will not be processed by our approach. This step is called data pre-processing in our approach. It is very similar to classification in Ref. [1] and architectural clue gathering in Ref. [24].

To detect the special methods, a thorough investigation is required. We are using an approach similar to that presented in Ref. [24], which takes full use of the static structure and naming conventions of the methods. The methods that will be viewed as special methods and not be processed are as follows [1, 24]: Constant method, Empty method, Getter and Setter method, Collection accessor, State method, Alias method, Factory methods, Delegation method.

4.4 Software dependency networks

After data pre-processing, two kinds of software dependency networks will be extracted and built: attribute-method networks and method-method networks. They serve as a basis for applying inspection and guided community detection algorithm to detect the refactorings. In this section, we will give the formal description of the two kinds of software dependency networks.

**Definition 1** (Attribute-Method Networks, AMN) In AMN the nodes represent the attributes or methods. Each attribute or method is represented by only one node respectively. Edges between two nodes denote one method accesses one attribute, i.e., if method A accesses attribute B,
there will be an edge between the node denoting method A and the node denoting attribute B. The direction of the edge will be ignored. Here, we only consider the presence of dependencies and neglect the multiplicity of dependencies such as method A depends three times on method B. See Fig. 2 for an example. Therefore, AMN can be described as:

\[ \text{Network}_{AMN} = (\text{Nodes, Edges}) \]  

(1)

where \( \text{Network}_{AMN} \) is an undirected network denoting AMN; \( \text{Nodes} \) denote methods or attributes; \( \text{Edges} \) denote the accessing relationships between attributes and methods.

**Definition 2** (Method-Method Networks, MMN) In MMN the nodes represent the methods in software systems. Each method is represented by only one node. The edge between two nodes denotes the dependency relationship between the two methods. Throughout the paper, we only take into consideration two kinds of method-method dependencies: Method-Method dependency through direct method call, and Method-Method dependency through accessing the same attribute. That is to say, if method A calls method B or method B calls method A, there will be an edge between nodes representing method A and B respectively; if method C accesses attribute attr, and method D accesses attr, there will also be an edge between the two nodes representing method C and D. The direction of the edge will be ignored. And here we also only consider the presence of dependencies and neglect the multiplicity of dependencies. Fig. 3 gives a simple example of MMN. Therefore, MMN can be described as:

\[ \text{Network}_{MMN} = (\text{Nodes, Edges}) \]  

(2)

where \( \text{Network}_{MMN} \) is the undirected network denoting MMN; \( \text{Nodes} \) denote methods; \( \text{Edges} \) denote the dependencies between methods.

### 4.5 Guided community detection algorithm

As is shown in Section 3.2, there are a lot of approaches that can be used to detect community structures in complex networks. A popular method now widely used relies on the optimization of a quantity called modularity, which, first proposed by Newman in Ref. [20], is a quality index for a partition of a network into communities. In this paper, we will also use this metric to evaluate the community structures in MMN.

However, recent research work reveal that modularity optimization may fail to identify communities smaller than a scale which depends on the total size of the network and on the degree of interconnectedness of the community, even in cases where communities are unambiguously defined [25]. To overcome this limit, we propose a guided community detection algorithm in this work. It starts from a state in which each method belongs to a specific community (not a random community as in Ref. [21]), the class where it defined, and proceeds by a series of method-moving operations at methods with dependencies to other methods not defined in the same classes. The rationale for this lies in the particularities in OO software systems: The class structures in OO software systems are the natural community structures existing in MMN. A majority of the methods are in the right classes, while only several misplaced methods lowering the quality of software systems need to be moved. Therefore, there is no need to start the community detection process in an unguided way with every method belonging to a random community as Ref. [21] does. Also, we can take a guided way with every method belonging to a natural community, the class where they are defined.

#### 4.5.1 Modularity evaluation metric

The metric used to evaluate the quality of one partition is of vital importance to our approach, for it controls the method-moving process. There are many different metrics that can be used to evaluate whether a particular community division is meaningful, such as \( MQ \) introduced by Mancoridis et al.
[26], EVM function of Tucker et al. [27] and modularity $Q$ devised by Newman [21] and Girvan. As talked above, in this paper, we use the quantitative definition proposed by Newman and Girvan. It is defined as:

$$Q = \sum_i (e_{ii} - a_i^2)$$

(3)

where $Q$ is the network modularity, $e_{ii}$ is the fraction of the edges that connect two nodes within community $i$, while $a_i$ is the fraction of edges that have at least one endpoint within community $i$.

After taking a method-moving operation we will recalculate the modularity $Q$ to decide whether to accept or reject this movement. Therefore, how to calculate $Q$ has a great impact on the performance of our algorithm even in cases where the number of methods is very large. In this paper, we will calculate $\Delta Q$, change in $Q$, rather than calculate $Q$. The change in $Q$ upon moving a method in community $i$ to community $j$ is given by:

$$\Delta Q = \Delta e_{ii} + \Delta e_{jj} - (\Delta a_{ii} + \Delta a_{jj})$$

(4)

where $\Delta e_{ii}$, $\Delta e_{jj}$, $\Delta a_{ii}$ and $\Delta a_{jj}$ are the differences of the corresponding values before and after one method in community $i$ is moved to community $j$.

In our approach, the algorithm travels through every method (node) with dependencies to other methods not defined in the same class (community), iteratively searches for the changes resulted from method-moving operation, and moves the method to the class that makes the largest increase in $Q$.

4.5.2 Guided community detection algorithm flow

Table 1 shows the algorithm flow of the guided community detection algorithm. In this algorithm, matrix CM$[i][j]$ is the node connection matrix, where CM$[i][j]=1$ means there is an edge between node $i$ and node $j$, otherwise there is no edge. $|Nodes|$ is the number of nodes in MMN. nodeId$[i]$ is an array storing the community identifier of node $i$, i.e., node $i$ belongs to community with identifier nodeId$[i]$. bvisited$[i]$ is an array with type bool denoting whether node $i$ has been visited or not. bSpecial$[i]$ is an array with type bool denoting whether node $i$ is a special method or not.

Table 1: Guided community detection algorithm flow

<table>
<thead>
<tr>
<th>Input</th>
<th>method-method networks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>optimized class structure and $Q$</td>
</tr>
</tbody>
</table>

1. Initialize CM$[i][j]$, nodeId$[i]$ (nodes in the same class will be assigned the same community identifier), bVisited$[i]$=false, and bSpecial$[i]$ (according to the results of data pre-processing).
2. Calculate $Q$ according to (3).
3. for(i=1; i<=|Nodes|; i++)//No.1
4. for(j=1; j<=|Nodes|; j++)//No.2
   if(CM$[i][j]$ & & !bVisited$[i]$ & & nodeId$[i]$=nodeId$[j]$ & & !bSpecial$[i]$)
   { //No.3
      bVisited$[i]$=true;
      Suppose move node $i$ to community nodeId$[j]$ and calculate $\Delta Q$, and store it into an array $\Delta Q[]$
   } //No.3
   } //No.2
5. Select the maximum $\Delta Q$, and move node $i$ to community with community identifier nodeId$[j]$ that produces the largest $\Delta Q$, and let node $j$ ($j=1,2,...,i$) who has an edge to node $i$ bVisited$[j]$=false; i=1;
6. Update $Q=\Delta Q+Q$;
7. //No.1

4.7 Inspection

Inspection is the process to travel through all the nodes in every AMN, and check whether there exist some methods which only access attributes with the type of another class. These methods are viewed as refactorings in our approach and should be moved to the class which is the type of these attributes. Because these methods introduce extra dependencies between the two classes, it increases the coupling. See Fig. 4 for an example, method1() and method2() in ClassB are viewed as refactorings and should be moved to ClassA, for they only access attributes with type ClassA. Moving them to ClassA can enhance the cohesion. The left part of Fig. 4 is the original code, and the right part is the code after
5.1 A practical case study

The proposed approach will be evaluated by the open-source Object-Oriented software system, JHotDraw 5.1\(^1\) which has been introduced in Refs. [1, 13] as a case study. It is a Java GUI framework for technical and structured graphics, developed by Gamma and Eggerschwiler, as a design exercise for using design patterns. There are two reasons for choosing JHotDraw 5.1 as a case study. On the one side, it is a well-known example for the use of design patterns and has a high quality. On the other side, we want to compare our approach with other existing approaches in the same case. Table 2 shows the statistic of JHotDraw 5.1. Where the number of classes, methods, and attributes are gathered from .class files, and others are gathered from the .java files. Details on how to get these data are provided in Section 5.2.

Table 2 JHotDraw 5.1 statistic

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total files</td>
<td>144</td>
<td>Blank lines</td>
<td>2448</td>
</tr>
<tr>
<td>Total size</td>
<td>144KB</td>
<td>Classes</td>
<td>172</td>
</tr>
<tr>
<td>Total lines</td>
<td>15430</td>
<td>Methods</td>
<td>1277</td>
</tr>
<tr>
<td>Code Lines</td>
<td>8419</td>
<td>Attributes</td>
<td>443</td>
</tr>
</tbody>
</table>

5.2 Experiment and results

In this section, we discuss how to extract the relevant networks and how our approach can be used for software refactoring. Our experiments were carried out on a PC at 2.53GHz with 1 GB of RAM.

The dependency networks and all data used in this paper are all automatically generated by our own developed software analysis tool named Software Network Analysis Tool (SNAT). It can process three kinds of files (.cpp, .java, .class), and extract the relevant information. However, it is currently not an open-source software system, so to help those who do not have such a tool to do class structure refactoring work, in this paper, we will show you how to use some open-source software systems to get the data you need indirectly.

Though there are many open-source analysis tools that can be used, such as CCC\(^2\), dependency finder\(^3\), ASM\(^4\), Doxygen\(^5\), and Max [28], AMNs and MMNs can not be generated directly by them. However, you can use them to assist your analysis. In the following sections, we will take dependency finder to show you how to get the needed data for the research work in this paper.

Dependency Finder is a powerful tool that can extract information from compiled Java code, including attributes, methods, classes, packages and dependencies between them. The obtained information is stored in a file with .xml file extension. The XML file is like a small database storing the information about the software analyzed. Thus, the only thing you should do is to parse the XML file to get the data you need. We have developed some small tools such as XmlParserForAMN and XmlParserForMMN (They are available for download\(^6\)) to parse the XML file to get the AMNs and MMNs which are stored in a file with .net file extension. You can take this .net file as the input for some social network analysis tools such as pajek\(^7\) and Ucinet [29] to visualize the dependency networks.

To get attribute-method networks, several steps should be taken. First, dependency finder can be used to extract the feature to feature dependency relationship data and stores them in a file with .xml file extension. The feature in dependency finder means attribute and method, and it makes no difference between the two. So when extracting the attribute-method networks, we should ignore those method call dependencies. Of course, we can use XmlParserForAMN to extract attribute-method networks, and use pajek to give visualizations.

---

1. JHotDraw 5.1. available at: http://www.jhotdraw.org/
2. CCC\(^\text{C}\). available at: http://ccc.sourceforge.net/
4. ASM. available at: http://asm.objectweb.org/
Table 3 shows the statistic of the attribute-method networks we got in the whole system. Fig. 5 gives an illustration of the attribute-method networks of the whole JHotDraw 5.1. The layout of the figure is set manually, not by the tool automatically.

Table 3 AMN statistic

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Attribute/method</td>
<td>957</td>
</tr>
<tr>
<td>Number of AMNs</td>
<td>136</td>
</tr>
<tr>
<td>Minimum AMN size</td>
<td>2</td>
</tr>
<tr>
<td>Maximum AMN size</td>
<td>49</td>
</tr>
</tbody>
</table>

Fig. 5 Attribute-method networks of the whole JHotDraw 5.1. Each color denotes a connected AMN.

Table 4 shows the refactorings that we get by inspection. The first column contains the methods proposed to be moved. The second column shows the suggested target classes. Here we omit the prefix CH.ifa.draw.util of the methods and target classes and the parameter list of methods. Fig. 6 gives an illustration of the AMN where these refactorings exist.

Table 4 Refactorings of AMNs

<table>
<thead>
<tr>
<th>Method</th>
<th>Target Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColorMap.Size</td>
<td>ColorEntry</td>
</tr>
<tr>
<td>ColorMap.static</td>
<td>ColorEntry</td>
</tr>
<tr>
<td>ColorMap.name</td>
<td>ColorEntry</td>
</tr>
<tr>
<td>ColorMap.colorIndex</td>
<td>ColorEntry</td>
</tr>
<tr>
<td>ColorMap.color</td>
<td>ColorEntry</td>
</tr>
</tbody>
</table>

Fig. 6 This is a small AMN. It contains the refactorings (these green nodes) we find by inspection.

The process to get method-method networks is similar to that of attribute-method networks. Of course, we can also use XmlParserForMMN to fulfill this work. Fig. 7 gives an illustration of the method-method networks of the whole JHotDraw 5.1. Here we only study the maximum connected network which is with the color of green in Fig. 7, for we found in experiments that the number of methods in other small connected networks contain less than 10, and all the methods in these small MMNs belong to the same class respectively. After data pre-processing, there are still 227 methods around in the maximum connected method-method network that can be processed by our approach.

Fig. 7 Method-method networks of the whole JHotDraw 5.1. Each color denotes a connected MMN.

There are only 5 methods (see Table 5 and Fig. 8) that were misplaced in the partition obtained after applying our guided community detection algorithm. The first column of Table 5 contains the name of methods proposed to be moved. The second column shows the suggested target classes. Here we omit the prefix CH.ifa.draw.util of the methods and target classes, together with the parameter list of methods.

Figure 9 is the change curve of the modularity with time.
over the course of our algorithm. Its maximum value is $Q=0.5281$, minimum value is 0.52724.

Table 5  Refactorings of MMNs

<table>
<thead>
<tr>
<th>Method</th>
<th>Target Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolygonFigure.chop</td>
<td>Geom</td>
</tr>
<tr>
<td>PertFigure.writeTasks</td>
<td>StorableOutput</td>
</tr>
<tr>
<td>PertFigure.readTasks</td>
<td>StorableInput</td>
</tr>
<tr>
<td>TextTool.fieldBounds</td>
<td>standard.TextHolder</td>
</tr>
<tr>
<td>TextTool.beginEdit</td>
<td>standard.TextHolder</td>
</tr>
</tbody>
</table>

Fig. 8  Illustration of the refactorings of MMNs. The labels beside the nodes are their names.

Fig. 9  The curve of modularity $Q$ vs. time over the course of the algorithm.

5.3 Analysis and comparison

Based on the analysis above, we can get 10 refactorings shown in Table 4 and Table 5. To judge whether the proposed refactorings make sense to the developer, we inspected these refactorings manually. By referring to the source files (.java) we can find that all proposed refactorings can be justified. For example, the methods shown in Table 4 are all suggested to be moved to class ColorEntry, for all these methods only access the array fMap with type ColorEntry. PolygonFigure.chop is suggested to be moved to the class Geom. As you can observe from the source code, it makes heavy use of the methods length2 and intersect defined in Geom but does not use any attributes or methods in its own class. From a metrics perspective moving PolygonFigure.chop to Geom reduces the coupling between the two classes. PertFigure.writeTasks, PertFigure.readTasks, TextTool.fieldBounds and TextTool.beginEdit can be similarly justified.

The advantages of using the proposed approach in comparison with the other approaches are illustrated below:

(1) the refactorings obtained by our approach is very similar to that reported in Refs. [1, 13];

(2) our approach provides a new perspective for software refactorings. It provides a new method to represent software system by dependency networks. It only uses community detection in dependency networks to find the meaningful refactorings. It makes software refactoring a simple work;

(3) the overall running time of the proposed guided community detection algorithm is less than 6 seconds, however the approach from Refs. [1, 13] are 30 minutes and 5 minutes, respectively. Here all executions were on a computer with a similar environment;

(4) our approach gives a list of meaningful refactorings that can be used in daily life of software engineering.

We cannot make a complete comparison with other approaches for software refactorings, for most of the proposed approaches, including Refs. [1, 13] only provide short examples indicating the refactorings they obtained. There is no way to access the relevant data and results.

6  Conclusion and future work

In this paper, we propose to represent software by two kinds of software dependency networks, and use a guided community detection algorithm to detect the refactorings and improve the class structure of a software system. Our approach can provide a list of refactorings that can be useful for assisting software engineers in their work of refactoring software systems.

Our major contribution to the software refactoring field is providing a new way to do software refactoring work from the perspective of dependency networks in software systems. We detail the theory to represent software systems by two kinds of dependency networks, describe a guided community detection algorithm for refactoring class structures, and evaluate our approach using an open-source software system.

Although our approach shows some feasibilities in refactoring class structures of JHotDraw 5.1, the broad validity of our approach demands further demonstration. Thus, the
future work include: (1) evaluating the approach using other open-source software systems including those using little design patterns; (2) implementing more refactorings to reveal more heuristics on high quality software systems; and (3) presenting a systematic software refactoring method which can refactor software systems at different levels of granularity such as method level, class level, package level and system level.

Acknowledgements We would like to thank the anonymous reviewers for their helpful comments. This work was supported by National Basic Research Program (973 Program) of China (2007CB310801), National High-Tech R & D Program (863 Program) of China (2006AA04Z156), National Natural Science Foundation of China (Grant Nos. 60873083, 60803025, 60703009 and 60303018), Natural Science foundation of Hubei Province (2008ABA358, 2008ABA379), Natural Science Foundation of Hubei Province for Distinguished Young Scholars (2008CDB351) and Research Fund for the Doctoral Program of Higher Education of China (20070486065).

References