Hindawi
Scientific Programming
Volume 2017, Article ID 4382348, 11 pages
https://doi.org/10.1155/2017/4382348

Research Article

Using Hierarchical Latent Dirichlet Allocation to Construct Feature Tree for Program Comprehension

Xiaobing Sun,[1] Xiangyue Liu,[2] Yucong Duan,[3] and Bin Li[1]

[1]*School of Information Engineering, Yangzhou University, Yangzhou, China*
[2]*Tongda College of Nanjing University of Posts and Telecommunications, Nanjing, China*
[3]*Hainan University, Haikou, China*

Correspondence should be addressed to Bin Li; lb@yzu.edu.cn

Received 9 November 2016; Revised 10 February 2017; Accepted 26 March 2017; Published 12 April 2017

Academic Editor: Michele Risi

Program comprehension is an important task faced by developers during software maintenance. With the increasing complexity of evolving systems, program comprehension becomes more and more difficult. In practice, programmers are accustomed to getting a general view of the features in a software system and then finding the interesting or necessary files to start the understanding process. Given a system, developers may need a general view of the system. The traditional view of a system is shown in a package-class structure which is difficult to understand, especially for large systems. In this article, we focus on understanding the system in both feature view and file structure view. This article proposes an approach to generate a feature tree based on hierarchical Latent Dirichlet Allocation (hLDA), which includes two hierarchies, the feature hierarchy and file structure hierarchy. The feature hierarchy shows the features from abstract level to detailed level, while the file structure hierarchy shows the classes from whole to part. Empirical results show that the feature tree can produce a view for the features and files, and the clustering of classes in the package in our approach is better (in terms of recall) than the other clustering approach, that is, hierarchical clustering.

1. Introduction

Understanding a software system at hand is one of the most frequently performed activities during software maintenance [1–3]. It is reported that developers working on software maintenance tasks spend up to 60% of their time for program comprehension [4–6]. Program comprehension is a process performed by a software practitioner using knowledge of both the semantics and syntax to build a mental model of its relation to the situation [7, 8]. As software evolves, its complexity usually increases as well [9]. Moreover, sometimes documents affiliated to the evolving system also become inaccessible or outdated, which makes the program comprehension activity even more difficult.

To reduce the difficulty of program comprehension, one of the effective approaches is to create a meaningful decomposition of large-scale system into smaller, more manageable subsystems, which is called software clustering [10–12]. A number of program comprehension techniques have been studied [13, 14]. The widely used clustering approaches are partitional clustering and hierarchical agglomerative clustering [15–17]. These approaches usually cluster the system based on static structural dependency in the program.

However, the results indicating the program decompositions based on the structure relationships of the software system can merely provide the structure view of these clusters. But for a program during software maintenance, the first activity faced by software developers is to locate the code that is relevant to the task (related to a functional feature) at hand [18]. Thus developers may be more interested in understanding the functional features of a system and how the source code corresponds to functional features. So some other clustering approaches are proposed based on semantic clustering, which exploits linguistic information in the source code identifiers and comments [14, 19–22]. All these clustering approaches provide either a structure view or a brief feature view for comprehension, but only a few can generate both of them.

In practice, understanding features from abstract level to detailed level and structure from whole to part can effectively help developers to understand a system in a stepwise manner. Such a whole-part way is more beneficial to developers when comprehending object oriented systems [3, 5]. For evolving software with increasing size and complexity, developers find it difficult to identify and choose the interesting packages and understand the chosen classes, their relationship, and their functionalities. Hence, a clustering representation considering both file structure and feature of the program should be constructed to ease program comprehension. With such clustering, developers have an easier, stepwise, and quicker understanding of the system.

In this article, we propose a feature tree to help understand a software system. Program comprehension is performed relying on two hierarchies of the functional features and file structure based on the feature tree. The feature tree contains the features from abstract level to detailed level and the file structure from whole to part. Hence, developers can obtain a good understanding of functional features of the whole system. The feature tree is generated based on hierarchical Latent Dirichlet Allocation (hLDA), which is a hierarchical topic model to analyze unstructured text [23, 24]. hLDA can be employed to discover a set of ideas or themes that well describe the entire text corpus in a hierarchical way. In addition, the model supports the assignment of the corresponding files to these themes, which are clusters for the software system corresponding to the functional features.

Therefore, our approach can be effectively used in a whole-part program comprehension way during software maintenance. Our approach is particularly suitable for researchers in the scientific and engineering computing area as researchers can employ our approach to understand the systems and maintain them in their own way. Researchers can easily understand these clusters for a software system. The main contributions of this article are as follows:

(1) We propose using hLDA to generate a feature tree. The tree includes feature hierarchy and file structure hierarchy. Feature hierarchy shows the features from abstract level to detailed level, while the file structure hierarchy shows the classes from whole to part.

(2) We provide a real case study to explain how the feature tree helps to understand the features and files for the JHotDraw program.

(3) We conduct empirical studies to show the effectiveness of our approach on two real-world open-source projects, JHotDraw and JDK.

The rest of the article is organized as follows: in the next section, we introduce the background of the hLDA model. Section 3 describes the details of our approach. Section 4 shows a real case study of the feature tree on JHotDraw program. We conduct empirical studies to validate the effectiveness of our approach in Section 5. The empirical results and threats to our studies are shown in Sections 6 and 7, respectively. In Section 8, related work using clustering for program comprehension is discussed. Finally, we conclude the article and outline directions for future work in Section 9.

2. Background

In this article, we use hLDA to cluster the classes in the software system into a hierarchical structure for easy program comprehension. This section discusses the background of hLDA.

Given an input corpus, which is a set of documents with each consisting of a sequence of words, hLDA is used to identify useful topics for the corpus and organize these topics into a hierarchy. In the hierarchy, more abstract topics are near its root and more concrete topics are near its leaves [23, 24]. hLDA is a model for multiple-topic documents which models dependency between topics in the documents from abstract to concrete. The model picks a topic according to its distribution and generates words according to the word distribution of the topic.

Let us consider a data set composed of a corpus of documents. Each document is a collection of words, where a word is an item in a vocabulary. Our basic assumption is that the topics in a document are generated according to a mixture model where the mixing proportions are random and document-specific. These topics are the basic mixture components in hLDA. The document-specific mixing proportions associated with these components are denoted by a vector Θ. We assume that there are K possible topics in the corpus and Θ is a K-dimensional vector. Suppose that we generate an L-level tree, where each node is associated with a topic. The process of applying hLDA is as follows:

(1) To select a path from the root to a leaf in the tree

(2) To generate a vector of topic proportions Θ with L-dimensional Dirichlet

(3) To identify the words in the document from the topics along the path from root to leaf based on the topic proportions Θ, w

Then, the nested CRP (Chinese Restaurant Process) is used to relax the assumption of a fixed tree structure [23, 24]. A document is then generated by first choosing an L-level path through the restaurants and then identifying the words from the L topics associated with the restaurants along that path. In this way, hLDA can generate a hierarchy where more abstract topics are near its root and more concrete topics are near its leaves.

3. Our Approach

Faced with the source code of a software system, developers need to use their domain knowledge to understand the features from abstract level to detailed level and the classes from whole to part. When understanding a system, developers first need to get a general understanding of the whole system and then find the interesting functions and source code. The process of our approach is shown in Figure 1. Firstly, the source code should be preprocessed for information retrieval technique; then the preprocessed corpus is processed with hLDA. Finally, we visualize the results in a feature tree view for program comprehension.

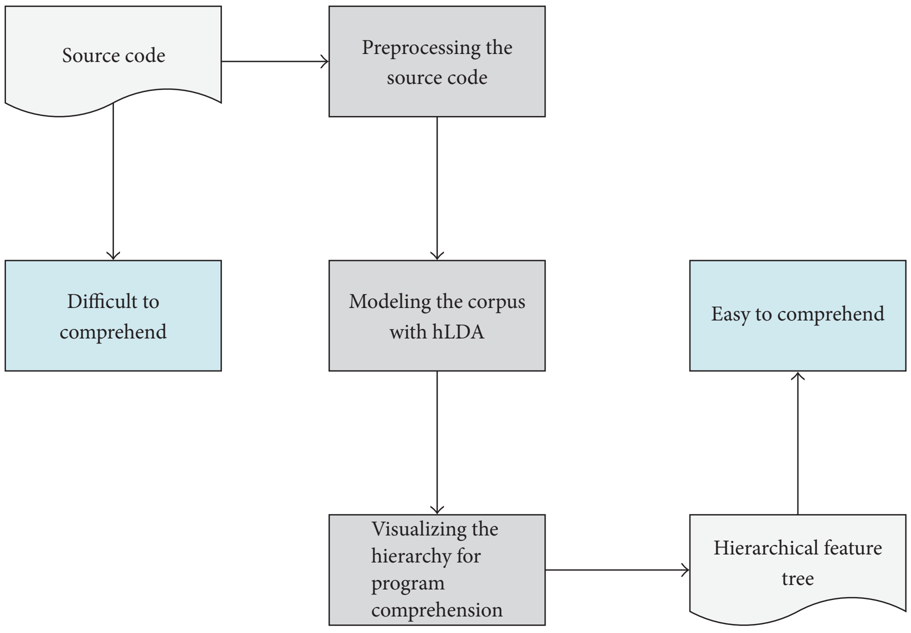

Source code
Preprocessing the source code
Difficult to comprehend
Modeling the corpus with hLDA
Easy to comprehend
Visualizing the hierarchy for program comprehension
Hierarchical feature tree

Figure 1: Process of our approach.

3.1. Preprocessing Source Code. We preprocess the source code of each software system by applying the typical source code preprocessing steps [25]. We firstly isolate source code identifiers and comments and then strip away syntax and programming language keywords. We deal with comments and identifiers differently due to their different programming requirements; that is, comments are natural language texts and identifiers are (composed of) single words.

The comments sometimes include authorship information which does not make sense in program comprehension. For example, words like *"author, distributed"* and other descriptions about the software itself are included in the header comments. So we first remove the header comments within the classes. Then, we follow four common operations to handle the remaining identifiers and comments as follows:

(1) *Tokenize*: we firstly tokenize each word in the source code to remove some numbers and punctuation marks

(2) *Split*: we split the identifiers based on common programming naming practices, for example, camel case *("oneTwo")* and underscores *("one_two")*

(3) *Remove stop words*: we remove common English language stop words *("in, it, for")* and key words *("int, return")* in the program to reduce the noise

(4) *Stem*: we stem the corpora to reduce the vocabulary size (e.g., *"changing"* becomes *"change"*)

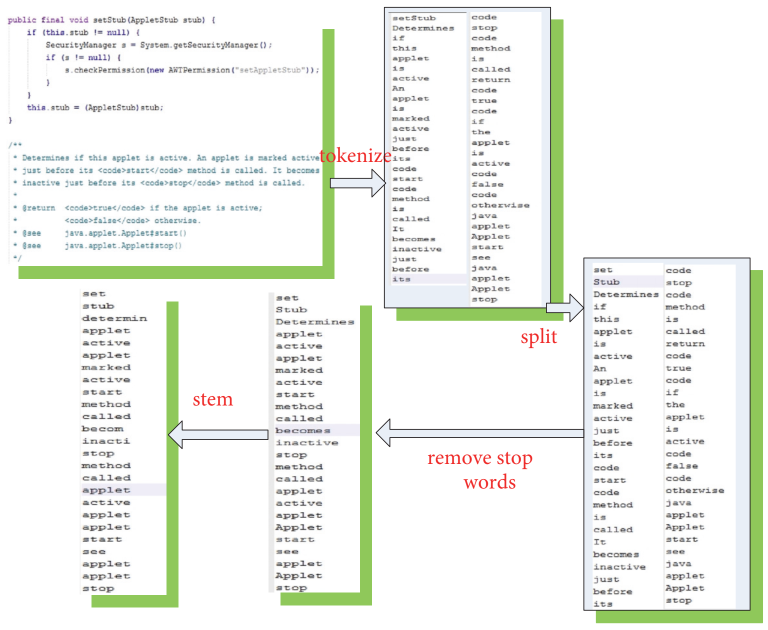
After these preprocessing operations on the unstructured source code, information retrieval can be more effectively used to extract the key information from the source code. Figure 2 shows an example of the process of preprocessing the source code in the class *Applet.java* in JDK. After tokenizing, splitting, removing the stop words, and stemming the source code, useful words are left for information retrieval application.

3.2. Modeling the Corpus with Hierarchical Latent Dirichlet Allocation. After preprocessing the source code of the software system, we apply hLDA to generate a hierarchy in which more abstract topics are near the root of the hierarchy and more concrete topics are near the leaves.

With the hLDA model, we can draw an outline about the topics, the files, and the hierarchical structure. The topics and the hierarchical structure constitute the feature hierarchy which shows the features from abstract level to detailed level. The files and the hierarchical structure constitute the file structure hierarchy, which shows the classes from whole to part.

3.3. Visualizing the Hierarchy for Program Comprehension. After modeling the corpus with hLDA, we can get a hierarchy of the topics for the software system. To get a view of the hierarchy, we display the hierarchy in a tree structure, which is called *feature tree*. The tree is generated from the results of the hLDA including two parts, one is the topic words and the other is the assigned classes. What is more, when comprehending the classes in each topic, developers may want to know the included package name. So we provide the package name when listing the relevant classes of each topic. In the *feature tree*, we can get three types of relationships: node to topic, father node to son node, and topic to file.

Node is an important part of the feature tree and it contains key information for program comprehension. The node consists of two elements, topics and relevant file names. Node in the first level generated by the hLDA model is the root of the feature tree which concludes the whole topics and files of the system. Nodes in the remaining levels are the



```
public final void setStub(AppletStub stub) {
    if (this.stub != null) {
        SecurityManager s = System.getSecurityManager();
        if (s != null) {
            s.checkPermission(new AWTPermission("setAppletStub"));
        }
    }
    this.stub = (AppletStub)stub;
}

/**
 * Determines if this applet is active. An applet is marked active
 * just before its <code>start</code> method is called. It becomes
 * inactive just before its <code>stop</code> method is called.
 *
 * @return  <code>true</code> if the applet is active;
 *          <code>false</code> otherwise.
 * @see     java.applet.Applet#start()
 * @see     java.applet.Applet#stop()
 */
```

```
setStub     code
Determines  stop
if          code
this        method
applet      is
is          called
active      return
An          code
applet      true
is          code
marked      if
active      the
just        applet
before      is
its         active
code        code
start       false
code        code
method      otherwise
is          java
called      applet
It          Applet
becomes     start
inactive    see
just        java
before      applet
its         Applet
            stop
```

```
set         code
Stub        stop
Determines  code
if          method
this        is
applet      called
is          return
active      code
An          true
applet      code
is          if
marked      the
active      applet
just        is
before      active
its         code
code        false
start       code
code        otherwise
method      java
is          applet
called      Applet
It          start
becomes     see
inactive    java
just        applet
before      Applet
its         stop
```

```
set
Stub
Determines
applet
active
applet
marked
active
start
method
called
becomes
inactive
stop
method
called
applet
active
applet
Applet
start
see
applet
Applet
stop
```

```
set
stub
determin
applet
active
applet
marked
active
start
method
called
becom
inacti
stop
method
called
applet
active
applet
applet
start
see
applet
applet
stop
```

Figure 2: An example of preprocessing the source code.

members of the subtree indicating the subtopics of the system and the related files. Topics are composed of words extracted from the preprocessed comments and identifiers based on the topic distribution in the hLDA model. So the topics are used for understanding the source code of interest. This is the feature hierarchy showing the features from abstract level to detailed level, which can help developers get a general view of the features of the software system. With a general view of the topics, users are more interested in the files corresponding to these topics. Relevant files are assigned based on the distribution of the hLDA with the generated topics. This is the other structure hierarchy showing the classes from whole to part in the form of clusters to show the classes corresponding to the features. In addition, when listing all the files, some more information is needed such as the location of the file. So the files and their locations are also presented in the tree nodes.

The structure of *feature tree* shows the relation between nodes. Father-child is the main relationship in the feature tree. All this information can be obtained from the results of the hLDA. Father node is a generalization of his son nodes. For the topics in the nodes, father node represents more general and abstract features than son nodes. For the files, father node includes all the files of his son nodes.

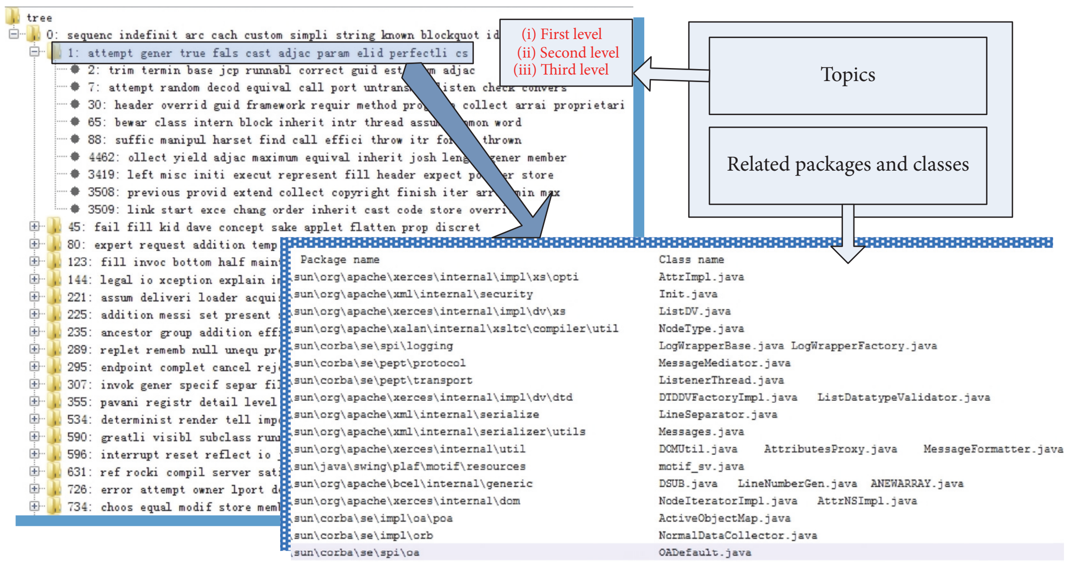
A three-level *feature tree* of the JDK program is displayed in Figure 3. We can see topics in each node with corresponding package names and class names in the content.

4. Case Study

In this section, we provide a case study of applying our approach to the JHotDraw program (this subject is also used in our empirical study. More about this subject will be introduced in Section 5.2). The JHotDraw program includes 23 packages and 305 classes.

We first generate the feature tree for the JHotDraw program. Part of the tree view is shown in Figure 4. The tree has three levels and there are several words describing each node in the tree. For the root node of the tree, 305 classes are included and the topic labeling the root node is *"invoc defin tool net delete modif preserve ad clone create draw."* Due to the step of preprocessing, some words are transformed into different forms. For the verbs in the topic, we can easily find that the original form of the verbs *"invoc defin delete modif preserve ad clone create draw"* is just *"invoke define delete modify preserve add clone create draw"* and these words express the functional features of the system. The nouns in the topic are *"tool net"* which indicate the objects of the system. JHotDraw defines a skeleton for GUI-based editor with tools in a tool palette, different views, user-defined graphical figures, and support for saving, loading, and so forth. So words in the root node describe the features of the software system to some extent.

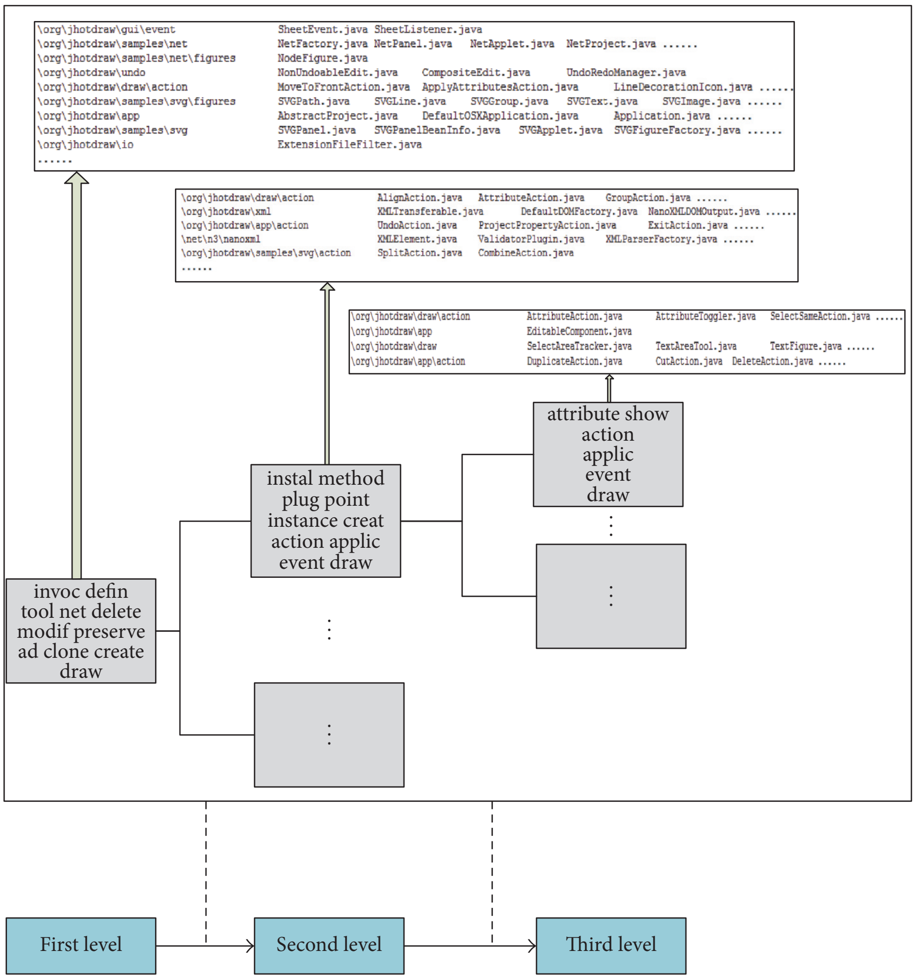
For the first son node in the second level of the root node, there are 163 classes. The node contains words like *"instal method plug point instance creat action applic event draw."* The original forms of these words are just *"install method plugin point instance create action application event draw."* The content to describe the functional features in this node is finer than these in the root node. For example, *"plugin application action event"* are words for plugin and application and some actions in the source code. As we know, a plugin program always includes *XML* language program. In the results, we can also find that packages like *"nanoxml, xml"* are in this node. Then, for the *"action"* feature, packages corresponding to it are distributed in this node like *"draw-action, samples-svg-action, app-action."* Some actions can be easily analyzed from the class names in the node such as *"SplitAction.java, CombineAction.java"* in *"samples-svg-action,"*



```
tree
 0: sequenc indefinit arc cach custom simpli string known blockquot id
   1: attempt gener true fals cast adjac param elid perfectli cs
     2: trim termin base jcp runnabl correct guid est   m adjac
     7: attempt random decod equival call port untrans   listen check convers
     30: header overrid guid framework requir method pro   collect arrai proprietari
     65: bewar class intern block inherit intr thread assu   mon word
     88: suffic manipul harset find call effici throw itr fo   thrown
     4462: ollect yield adjac maximum equival inherit josh leng   gener member
     3419: left misc initi execut represent fill header expect po   er store
     3508: previous provid extend collect copyright finish iter arr   min max
     3509: link start exce chang order inherit cast code store overri
   45: fail fill kid dave concept sake applet flatten prop discret
   80: expert request addition temp
   123: fill invoc bottom half maint
   144: legal io xception explain in
   221: assum deliveri loader acqui
   225: addition messi set present
   235: ancestor group addition eff
   289: replet rememb null unequ pr
   295: endpoint complet cancel rej
   307: invok gener specif separ fi
   355: pavani registr detail level
   534: determinist render tell imp
   590: greatli visibl subclass run
   596: interrupt reset reflect io
   631: ref rocki compil server sat
   726: error attempt owner lport d
   734: choos equal modif store mem
```

```
Package name                                   Class name
sun\org\apache\xerces\internal\impl\xs\opti    AttrImpl.java
sun\org\apache\xml\internal\security           Init.java
sun\org\apache\xerces\internal\impl\dv\xs      ListDV.java
sun\org\apache\xalan\internal\xsltc\compiler\util   NodeType.java
sun\corba\se\spi\logging                       LogWrapperBase.java LogWrapperFactory.java
sun\corba\se\pept\protocol                     MessageMediator.java
sun\corba\se\pept\transport                    ListenerThread.java
sun\org\apache\xerces\internal\impl\dv\dtd     DTDDVFactoryImpl.java   ListDatatypeValidator.java
sun\org\apache\xml\internal\serialize          LineSeparator.java
sun\org\apache\xml\internal\serializer\utils   Messages.java
sun\org\apache\xerces\internal\util            DOMUtil.java    AttributesProxy.java    MessageFormatter.java
sun\java\swing\plaf\motif\resources            motif_sv.java
sun\org\apache\bcel\internal\generic           DSUB.java   LineNumberGen.java  ANEWARRAY.java
sun\org\apache\xerces\internal\dom             NodeIteratorImpl.java   AttrNSImpl.java
sun\corba\se\impl\oa\poa                       ActiveObjectMap.java
sun\corba\se\impl\orb                          NormalDataCollector.java
sun\corba\se\spi\oa                            OADefault.java
```

FIGURE 3: The feature tree of the JDK program.



```
\org\jhotdraw\gui\event                SheetEvent.java SheetListener.java
\org\jhotdraw\samples\net              NetFactory.java NetPanel.java   NetApplet.java  NetProject.java ......
\org\jhotdraw\samples\net\figures      NodeFigure.java
\org\jhotdraw\undo                     NonUndoableEdit.java    CompositeEdit.java      UndoRedoManager.java
\org\jhotdraw\draw\action              MoveToFrontAction.java  ApplyAttributesAction.java      LineDecorationIcon.java ......
\org\jhotdraw\samples\svg\figures      SVGPath.java    SVGLine.java    SVGGroup.java   SVGText.java    SVGImage.java ......
\org\jhotdraw\app                      AbstractProject.java    DefaultOSXApplication.java      Application.java ......
\org\jhotdraw\samples\svg              SVGPanel.java   SVGPanelBeanInfo.java   SVGApplet.java  SVGFigureFactory.java ......
\org\jhotdraw\io                       ExtensionFileFilter.java
......
```

```
\org\jhotdraw\draw\action              AlignAction.java   AttributeAction.java    GroupAction.java ......
\org\jhotdraw\xml                      XMLTransferable.java       DefaultDOMFactory.java  NanoXMLDOMOutput.java ......
\org\jhotdraw\app\action               UndoAction.java    ProjectPropertyAction.java      ExitAction.java ......
\net\n3\nanoxml                        XMLElement.java    ValidatorPlugin.java    XMLParserFactory.java ......
\org\jhotdraw\samples\svg\action       SplitAction.java   CombineAction.java
......
```

```
\org\jhotdraw\draw\action              AttributeAction.java      AttributeToggler.java   SelectSameAction.java ......
\org\jhotdraw\app                      EditableComponent.java
\org\jhotdraw\draw                     SelectAreaTracker.java    TextAreaTool.java       TextFigure.java ......
\org\jhotdraw\app\action               DuplicateAction.java      CutAction.java  DeleteAction.java ......
```

invoc defin tool net delete modif preserve ad clone create draw
instal method plug point instance creat action applic event draw
attribute show action applic event draw
⋮
⋮
⋮
⋮
First level
Second level
Third level

FIGURE 4: A part of the overview of the feature tree for JHotDraw.

which indicates the actions of split and combine for the SVG (Scalable Vector Graphics).

For the first node in the third level, which is the son node of the node mentioned above, 26 classes are assigned to this node. The labeled topic is *“attribute show action applic event draw.”* The original form of these words is just *“attribute show action application event draw.”* Most of them have appeared in their father node. The representative word is *“attribute.”* In JHotDraw, *AttributeFigure* is a directly accessed component. We can further refine its behavior by overriding some methods such as *draw()* to customize the graphical representation in the diagram. The node only includes four packages. They are *draw.action, app, draw, app.action*. In *“draw.action,”* we may find *“AttributeAction.java, AttributeToggler.java, SelectSameAction.java, DefaultAttributeAction.java,”* which correspond to the word *“attribute.”*

Hence, from the above study on a real software system, we can use the information on the feature tree to facilitate understanding the functional features and file structure in the system.

5. Empirical Study

In this section, we conduct empirical studies to evaluate the effectiveness of our approach. In our studies, we address the following two research questions:

(RQ1) Does the feature hierarchy really help users to comprehend the software system?

(RQ2) Are the clustering results more convenient for users to understand the software system than the hierarchical clustering approach [16]?

(RQ1) and (RQ2) are used to evaluate the feature hierarchy and file structure hierarchy, respectively. (RQ1) indicates whether the results of the feature tree really help users to comprehend the software system. In addition, we investigate (RQ2) to see whether the clustering results can make it easy for users to comprehend software systems compared with the results of hierarchical clustering [16].

5.1. Subject Systems. We address our research questions by performing studies on the source code of two well-known software systems, JHotDraw (https://sourceforge.net/projects/jhotdraw), and two packages in the Java Development Kit 1.7 (http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html), that is, JDK-sun and JDK-java. JHotDraw is a medium-sized, 2D drawing framework developed in Java. The Java Development Kit (JDK) is implementation of either one of the Java SE or Java EE platforms in the form of a binary product. Specifics of these subject systems are shown in Table 1.

These projects belong to different problem domains with medium or large size. They are all real-world software systems and have been widely used as empirical study in the context of software maintenance or program comprehension [26, 27].

Table 1: Subject systems.

Subject	File	Package	Class	KLoC
JHotDraw	144	23	305	31
JDK-sun	1631	58	1604	408
JDK-java	3221	260	2988	225

5.2. Design of Our Study. In our approach, there is a parameter, that is, tree level L. It represents the level of the feature tree, which affects the tree structure for program comprehension and the clustering results in the software system, respectively. In our study, we consider L to be 3 for manual program understanding since the level of three achieves relatively good results in our study.

For the hLDA computation, we used the C++ implication from David Blei’s hLDA topic modeling software (https://github.com/xch91/hlda-cpp). We ran it for 10,000 sampling iterations, the first 1000 of which were used for parameter optimization.

In addition, we used hierarchical clustering algorithm for a comparative study. Anquetil and Lethbridge proposed a hierarchical clustering algorithm suite, which provides a selection of association and distance coefficients as well as update rules to cluster software systems [16]. There are three variants of *hierarchical agglomerative clustering*, which were used in many software architecture recovery techniques [28, 29]. These variants can be distinguished based on their update rules as follows: Single Linkage (SL), Complete Linkage (CL), and Average Linkage (AL). SL merges two clusters with the smallest minimum pairwise distance. CL merges two clusters with the largest minimum pairwise distance and AL merges two clusters with the average minimum pairwise distance. Previous works have shown that, in relatively large systems, CL generates the best results in terms of recovering authoritative decompositions [16, 30, 31]. We also used the same semantic data (words in comments and identifiers) to perform the *hierarchical agglomerative clustering*.

5.3. Participants. We conducted a user study to answer the two research questions. Our study involved 10 participants from school and/or industry. These participants have different levels of software development experience and familiarity with Eclipse. Half of them are from our university (graduates in our lab) with 2-3 years of development experience and the other half are from industry with 5-6 years of development experience, especially large project development experience. They were required to conduct program comprehension for a system (e.g., JDK-java, JDK-com, and JHotDraw). In addition, we gave participants the classes, and they needed to identify the relevant classes (semantic relevant) for them.

5.4. Measures. For (RQ1), we investigated whether the feature hierarchy helps the participants comprehend the functional features of whole system. To show whether the topic hierarchy generated by our approach is useful, the participants needed to assess whether the feature hierarchy enables them to understand the clusters. A five-point Likert scale with 1 (very useless) to 5 (very useful) assesses each participant’s view of the topic hierarchy. According to the scores, we can see whether the structure helps program comprehension.

To answer (RQ2), we used precision and recall, two widely used metrics for information retrieval and clustering evaluation [32, 33], to evaluate the accuracy of the topics and the clustering results. Precision in clustering evaluation is the percentage of intrapairs proposed by the clustering approach, which are also intrapairs in the authoritative decomposition. Recall is the percentage of intrapairs in the authoritative decomposition, which are also intrapairs in the decomposition proposed by the clustering approach. The authoritative terms are given by each participant. We found that the participants gave different results for the given classes. So we consider that a class is included in the authoritative decomposition when 60% of participants or more selected it as relevant one. These two metrics are used to measure the accuracy of the clustering results in an a posteriori way. They are defined as follows:

$$\text{precision} = \frac{|\text{authoritative cluster} \cap \text{estimated cluster}|}{|\text{estimated cluster}|}$$
$$\text{recall} = \frac{|\text{authoritative cluster} \cap \text{estimated cluster}|}{|\text{authoritative cluster}|}. \quad (1)$$

To avoid optimizing for either precision or recall, F-measure is used, which is the harmonic mean of precision and recall. F is the harmonic mean of precision and recall. It is defined as follows.

$$F = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}. \quad (2)$$

Since agglomerative clustering can generally be adjusted to increase or decrease recall at the expense of precision by changing parameters, we selected 50 results with the best F values and then computed the average values for our comparative study.

5.5. Variables. To perform our study, the main independent variable of the empirical study is the clustering techniques (hLDA and CL) that we used to generate the clustering results for program comprehension. The dependent variables are the values of precision (P), recall (R), and F (F), which are used to measure the accuracy of these clustering techniques.

6. Empirical Results

In this section, we collect and discuss the results from our studies to answer the proposed two research questions, respectively.

6.1. (RQ1). Our approach is aimed at giving a view of the topics in a hierarchical way from a general level to a concrete level for the whole system and provides a clustering result of the packages and classes. In this subsection, we discuss whether the feature tree structure of the topics helps to understand the whole system and whether the topics are representative.

We provided the topic tree generated from hLDA of each system to the participants to investigate whether the tree can give a view of the topics in a hierarchical way from a general level to a concrete level for program comprehension. First, we provided the topics of each cluster to the participants. They used a five-point Likert scale to answer questions related to the quality of the topics. The results are shown in Table 2. The results show that the average score of the results is around 3.5, which indicates that the participants think that the topics are useful to understand the cluster. So, for program comprehension, some topics labeling the clusters are useful to help users to understand the program.

Table 2: The score for the quality of the topics.

Subject	Participants										AVG
	$P1$	$P2$	$P3$	$P4$	$P5$	$P6$	$P7$	$P8$	$P9$	$P10$	
JHotDraw	4	4	4	3	4	3	4	4	4	3	3.7
JDK-sun	3	3	4	3	3	3	4	3	4	3	3.3
JDK-java	4	4	3	3	3	3	4	3	4	3	3.4

Table 3: Precision and recall of two clustering approaches for each subject system.

Subject	CL			hLDA		
	P	R	F	P	R	F
JHotDraw	53%	23%	32%	21%	40%	28%
JDK-sun	60%	25%	35%	40%	38%	39%
JDK-java	48%	14%	22%	32%	42%	36%
AVG	54%	20%	29%	31%	40%	35%

6.2. (RQ2). In this subsection, we compare the accuracy of the two clustering results in leaves of the feature tree generated by our approach and the hierarchical CL clustering approach.

To quantitatively compare these two clustering approaches, we compute their precision and recall results, which are shown in Table 3. From the results, we notice that the average precision of CL is 54% and the highest is 60% for JDK-sun. And all the precision values of the three subject programs are higher than the hLDA approach.

But, for the recall results, we notice that the average recall of hLDA is around 40% and the highest is 42% for JDK-java. And all the recall values of the three subject programs are higher than CL, which means that the hLDA model can predict more files for the clusters. This is because the hLDA model clusters the files or classes not only with the same words but also with the same latent topics. That is to say, our approach can cover more relevant classes in the authoritative clusters, which can effectively facilitate program comprehension.

Although clustering results with high precision are more correct, many other correct relevant results are not discovered [16]. When using clustering for program comprehension, high recall is more important [16]. So, from the results discussed above, our approach can effectively identify more relevant classes in a cluster to help program comprehension compared with the CL approach.

7. Threats to Validity

Like any empirical validation, ours has its limitations. In the following, threats to the validity of our studies are discussed.

7.1. Threats to External Validity. We only applied our technique to two subject programs. Thus we cannot guarantee that the results in our studies can be generalized to other more complex or arbitrary subjects. However, these subjects are selected from open-source projects and widely employed for experimental studies [26, 27]. In addition, we only selected part of reprehensive subjects and used them for comparison with the CL clustering approach when evaluating the effectiveness of clustering results.

In addition, we considered only Java programming language and Eclipse development environment. Further studies are required to generalize our findings in large-scale industrial projects and with developers who have sufficient domain knowledge and familiarity with the subject systems in various development environments.

7.2. Threats to Internal Validity. First, the preprocessing of the program is to transform the source code into word list, which is then used for information retrieval. This may affect the results of the hLDA model. Since we derive our topics based on the use of identifier names and comments, the quality of the topics generated by hLDA relies on the quality of the source code preprocessing.

Another threat is just like other comprehension techniques based on semantics demonstrating that the code should have a high quality in both name rules and the comments. Some programs without high quality may not get good results as in our study.

A third threat to the internal validity of our study is the difference in the capabilities of the participants. Specifically, we cannot eliminate the threat that the participants have different capabilities. This also can affect the accuracy of the results. As the authoritative decomposition changes, the results may also be different.

7.3. Threats to Construct Validity. To evaluate the effectiveness of our approach, we used precision and recall metrics. These two metrics only focused on the false-positives and false-negatives for authoritative clustering results. However, for program comprehension, other factors may be more important.

In addition, when we compare our approach with agglomerative clustering, we selected 50 results with the best F values and then computed the average values for our comparative study. However, agglomerative clustering approaches can generally be adjusted to increase or decrease recall at the expense of precision by changing parameters. Some other design approaches may obtain different results.

8. Related Work

Source code is one of the important inputs for program comprehension. There are a number of studies focusing on this area [10, 34–38]. Program clustering is one of the effective ways for program comprehension. There are two different types of program clustering, one is based on syntactic dependency analysis [39–43], while the other is based on the semantic information analysis [44, 45].

The syntactic based clustering approaches usually focus on analyzing the structural relationship among entities, for example, call dependence, control, and data dependence. Mancoridis et al. proposed an approach which generates clusters using module dependency graph for the software system [36]. Anquetil and Lethbridge proposed an approach, which generates clusters using weighted dependency graph for program clustering [46]. Sartipi and Kontogiannis presented an interactive approach to recovery cohesive subsystems within C systems. They analyze different relationships and build an attributed relational graph. Then the graph is manually or automatically partitioned using data mining techniques for program clustering [47]. For all these works, they analyzed syntactic relationships to cluster the program, and developers understand how the functional features are programmed in the source code based on this syntactic clustering. In this article, we focus on extracting the functional features in the source code and clustering the source code based on hLDA.

Semantic based clustering approaches just attempt to analyze the functional features in a system [13, 48]. The functional features in the source code are extracted from comments, identifier names, and file names [49]. For example, Kuhn et al. proposed an approach to group software artifacts based on Latent Semantic Indexing. They focused on grouping source code containing similar terms in the comments [14, 50]. Scanniello et al. presented an approach which also uses Latent Semantic Indexing to get the dissimilarity between the entities [51]. Corazza et al. introduce the concept of zones (e.g., comments and class names) in the source code to assign different importance to the information extracted from different zones [50, 52]. They use a probabilistic model to automatically give weights to these zones and apply the Expectation Maximization (EM) algorithm to derive values for these weights. In this article, our approach used hLDA to generate a feature tree structure of the topic hierarchy for program comprehension and cluster the packages and classes based on them. The feature tree includes two hierarchies for the functional features and clusters of packages and classes.

In addition, some other approaches combine the syntactic analysis and semantic analysis for program clustering [11, 53–55]. Tzerpos proposed an ACDC algorithm, which uses name and dependency of classes to cluster all the system into small clusters for comprehension [56]. Adritsos et al. presented an approach, LIMBO, which considers both structural and nonstructural attributes to decompose a system into clusters, while in this article we used only the semantic analysis for clustering. But our approach can generate topics to help users more easily understand the classes or packages in each cluster.

9. Conclusion and Future Work

In this article, we proposed an approach of generating a feature tree based on hLDA, which includes two hierarchies for functional features and file structure. The feature hierarchy shows the features from abstract level to detailed level, while the file structure hierarchy shows the classes from whole to part. We conducted empirical studies to show the effectiveness of our approach on two real-world open-source projects. The results show that the results of our approach are more effective than the hierarchical CL clustering approach. In addition, the topics labeling these clusters are useful to help developers understand them. Therefore, our approach could provide an effective way for developers to understand the system quickly and accurately.

In our study, we only conducted our studies on two Java-based programs, which cannot imply its generality for other types of systems. Future work will focus on conducting more studies on different types of systems to evaluate the generality of our approach. In addition, we find that, just like other IR approaches, the preprocessing process, the scale of the corpus, and the parameters in the model can affect the results sensitively. Further study on how to make the best use of IR approach for programs transformation is necessary.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work is supported partially by National Natural Science Foundation of China under Grant no. 61402396, no. 61472344, no. 61662021, and no. 61602267, partially by the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University under Grant no. KFKT2016B21, partially by the Jiangsu Qin Lan Project, partially by the China Postdoctoral Science Foundation under Grant no. 2015M571489, and partially by the Natural Science Foundation of the Jiangsu Higher Education Institutions of China under Grant no. 15KJB520030.

References

[1] T. Fritz, G. C. Murphy, E. Murphy-Hill, J. Ou, and E. Hill, "Degree-of-knowledge: modeling a developer's knowledge of code," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 2, article 14, 2014.

[2] Z. Soh, "Context and vision: studying two factors impacting program comprehension," in *Proceedings of the IEEE 19th International Conference on Program Comprehension (ICPC '11)*, pp. 258–261, June 2011.

[3] J.-M. Burkhardt, F. Détienne, and S. Wiedenbeck, "Object-oriented program comprehension: effect of expertise, task and phase," *Empirical Software Engineering*, vol. 7, no. 2, pp. 115–156, 2002.

[4] T. Nakagawa, Y. Kamei, H. Uwano, A. Monden, K. Matsumoto, and D. M. German, "Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: a controlled experiment," in *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, pp. 448–451, June 2014.

[5] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 4, pp. 31:1–31:37, 2014.

[6] Y. Kong, M. Zhang, and D. Ye, "A belief propagation-based method for task allocation in open and dynamic cloud environments," *Knowledge-Based Systems*, vol. 115, pp. 123–132, 2017.

[7] M. P. O'Brien, "Software comprehension: a review and research direction," Tech. Rep., Department of Computer Science & Information Systems, University of Limerick, Limerick, Ireland, 2003.

[8] T. Kosar, M. Mernik, and J. C. Carver, "Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments," *Empirical Software Engineering*, vol. 17, no. 3, pp. 276–304, 2012.

[9] K. Maruyama, T. Omori, and S. Hayashi, "A visualization tool recording historical data of program comprehension tasks," in *Proceedings of the 22nd International Conference on Program Comprehension (ICPC '14)*, pp. 207–211, June 2014.

[10] C. Y. Chong, S. P. Lee, and T. C. Ling, "Efficient software clustering technique using an adaptive and preventive dendrogram cutting approach," *Information and Software Technology*, vol. 55, no. 11, pp. 1994–2012, 2013.

[11] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 150–165, 2005.

[12] M. Bauer and M. Trifu, "Architecture-aware adaptive clustering of OO systems," in *Proceedings of the European Conference on Software Maintainance and Reengineering (CSMR '04)*, pp. 3–14, Tampere, Finland, March 2004.

[13] G. Santos, M. T. Valente, and N. Anquetil, "Remodularization analysis using semantic clustering," in *Proceedings of the Software Evolution Week—IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE '14)*, pp. 224–233, Antwerp, Belgium, February 2014.

[14] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[15] C. Li, Z. Xu, C. Qiao, and T. Luo, "Hierarchical clustering driven by cognitive features," *Science China. Information Sciences*, vol. 57, no. 1, 012109, 14 pages, 2014.

[16] N. Anquetil and T. C. Lethbridge, "Experiments with clustering as a software remodularization method," in *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE '99)*, pp. 235–255, October 1999.

[17] Z. Fu, K. Ren, J. Shu, X. Sun, and F. Huang, "Enabling personalized search over encrypted outsourced data with efficiency improvement," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2546–2559, 2016.

[18] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[19] G. Santos, M. T. Valente, and N. Anquetil, "Remodularization analysis using semantic clustering," in *Proceedings of the 1st Software Evolution Week—IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE '14)*, pp. 224–233, Antwerp, Belgium, February 2014.

[20] B. L. Vinz and L. H. Etzkorn, "Improving program comprehension by combining code understanding with comment understanding," *Knowledge-Based Systems*, vol. 21, no. 8, pp. 813–825, 2008.

[21] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 800–813, 1991.

[22] Z. Fu, X. Wu, C. Guan, X. Sun, and K. Ren, "Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 12, pp. 2706–2716, 2016.

[23] D. M. Blei, T. L. Griffiths, and M. I. Jordan, "The nested Chinese restaurant process and Bayesian nonparametric inference of topic hierarchies," *Journal of the ACM*, vol. 57, no. 2, article 7, 2010.

[24] D. M. Blei, T. L. Griffiths, M. I. Jordan, and J. B. Tenenbaum, "Hierarchical topic models and the nested chinese restaurant process," in *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8–13, 2003, Vancouver and Whistler, British Columbia, Canada]*, pp. 17–24, 2003.

[25] X. Sun, X. Liu, J. Hu, and J. Zhu, "Empirical studies on the NLP techniques for source code data preprocessing," in *Proceedings of the 3rd International Workshop on Evidential Assessment of Software Technologies (EAST '14)*, pp. 32–39, May 2014.

[26] U. Erdemir, U. Tekin, and F. Buzluca, "Object oriented software clustering based on community structure," in *Proceedings of the 18th Asia Pacific Software Engineering Conference (APSEC '11)*, pp. 315–321, IEEE, Ho Chi Minh, Vietnam, December 2011.

[27] A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR methods for labeling source code artifacts: is it worthwhile?" in *Proceedings of the IEEE 20th International Conference on Program Comprehension (ICPC '12)*, pp. 193–202, Passau, Germany, June 2012.

[28] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, 2007.

[29] O. Maqbool and H. A. Babri, "The weighted combined algorithm: a linkage algorithm for software clustering," in *Proceedings of the European Conference on Software Maintainance and Reengineering (CSMR '04)*, pp. 15–24, Tampere, Finland, March 2004.

[30] M. Shtern and V. Tzerpos, "Clustering methodologies for software engineering," *Advances in Software Engineering*, vol. 2012, Article ID 792024, 18 pages, 2012.

[31] A. Mahmoud and N. Niu, "Evaluating software clustering algorithms in the context of program comprehension," in *Proceedings of the 21st International Conference on Program Comprehension (ICPC '13)*, pp. 162–171, May 2013.

[32] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 1, Article ID 2559935, 4 pages, 2014.

[33] C. J. van Rijsbergen, *Information Retrieval*, Butterworths, London, UK, 1979.

[34] X. Liu, X. Sun, B. Li, and J. Zhu, "PFN: a novel program feature network for program comprehension," in *Proceedings of the IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS '14)*, pp. 349–354, Taiyuan, China, June 2014.

[35] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proceedings of the 10th International Workshop on Program Comprehension (IWPC '02)*, pp. 271–278, June 2002.

[36] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *Proceedings of the 6th International Workshop on Program Comprehension (IWPC '98)*, p. 45, Ischia, Italy, June 1998.

[37] D. Binkley, D. Heinz, D. J. Lawrie, and J. Overfelt, "Understanding LDA in source code analysis," in *Proceedings of the 22nd International Conference on Program Comprehension (ICPC '14)*, pp. 26–36, ACM, Hyderabad, India, June 2014.

[38] B. Gu, V. S. Sheng, K. Y. Tay, W. Romano, and S. Li, "Incremental support vector learning for ordinal regression," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 26, no. 7, pp. 1403–1416, 2015.

[39] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[40] S. Islam, J. Krinke, D. Binkley, and M. Harman, "Coherent clusters in source code," *Journal of Systems and Software*, vol. 88, no. 1, pp. 1–24, 2014.

[41] S. Mirarab, A. Hassouna, and L. Tahvildari, "Using Bayesian belief networks to predict change propagation in software systems," in *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC '07)*, pp. 177–186, Alberta, Canada, June 2007.

[42] F. Deng and J. A. Jones, "Weighted system dependence graph," in *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST '12)*, pp. 380–389, Montreal, Canada, April 2012.

[43] M. Gethers, A. Aryani, and D. Poshyvanyk, "Combining conceptual and domain-based couplings to detect database and code dependencies," in *Proceedings of the IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM '12)*, pp. 144–153, Riva del Garda, Italy, September 2012.

[44] L. Guerrouj, "Normalizing source code vocabulary to support program comprehension and software quality," in *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*, pp. 1385–1388, IEEE, San Francisco, Calif, USA, May 2013.

[45] A. De Lucia, M. Di Penta, and R. Oliveto, "Improving source code lexicon via traceability and information retrieval," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 205–227, 2011.

[46] N. Anquetil and T. C. Lethbridge, "Recovering software architecture from the names of source files," *Journal of Software Maintenance and Evolution*, vol. 11, no. 3, pp. 201–221, 1999.

[47] K. Sartipi and K. Kontogiannis, "A user-assisted approach to component clustering," *Journal of Software Maintenance and Evolution*, vol. 15, no. 4, pp. 265–295, 2003.

[48] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "MUDABlue: an automatic categorization system for open source repositories," in *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC '04)*, pp. 184–193, Busan, Korea, December 2004.

[49] A. Kuhn, S. Ducasse, and T. Gîrba, "Enriching reverse engineering with semantic clustering," in *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE '05)*, pp. 133–142, Pittsburgh, Pa, USA, November 2005.

[50] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR '11)*, pp. 35–44, IEEE, Oldenburg, Germany, March 2011.

[51] G. Scanniello, M. Risi, and G. Tortora, "Architecture recovery using Latent Semantic Indexing and k-Means: an empirical evaluation," in *Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM '10)*, pp. 103–112, September 2010.

[52] A. Corazza, S. Di Martino, and G. Scanniello, "A probabilistic based approach towards software system clustering," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR '10)*, pp. 88–96, Madrid, Spain, March 2010.

[53] G. Scanniello, A. D'Amico, C. D'Amico, and T. D'Amico, "Using the Kleinberg algorithm and vector space model for software system clustering," in *Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC '10)*, pp. 180–189, Minho, Portugal, July 2010.

[54] G. Scanniello and A. Marcus, "Clustering support for static concept location in source code," in *Proceedings of the IEEE 19th International Conference on Program Comprehension (ICPC '11)*, pp. 1–10, Kingston, Canada, June 2011.

[55] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *Proceedings of the 23rd International Conference ojn Software Engineering (ICSE '01)*, pp. 103–112, Toronto, Canada, May 2001.

[56] V. Tzerpos and R. Holt, "ACCD: an algorithm for comprehension-driven clustering," in *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE '00)*, pp. 258–267, Brisbane, Australia, November 2000.