Modeling the Ownership of Source Code Topics

Christopher S. Corley, Elizabeth A. Kammer, Nicholas A. Kraft
Department of Computer Science
The University of Alabama
Tuscaloosa, AL 35487-0290
{cscorley,eakammer}@ua.edu, nkraft@cs.ua.edu

Abstract—**Exploring linguistic topics in source code is a program comprehension activity that shows promise in helping a developer to become familiar with an unfamiliar software system. Examining ownership in source code can reveal complementary information, such as who to contact with questions regarding a source code entity, but the relationship between linguistic topics and ownership is an unexplored area. In this paper we combine software repository mining and topic modeling to measure the ownership of linguistic topics in source code. We conduct an exploratory study of the relationship between linguistic topics and ownership in source code using 10 open source Java systems. We find that classes that belong to the same linguistic topic tend to have similar ownership characteristics, which suggests that conceptually related classes often share the same owner(s). We also find that similar topics tend to share the same ownership characteristics, which suggests that the same developers own related topics.**

Index Terms—**program comprehension, mining software repositories, ownership, topic modeling, pachinko allocation model.**

I. Introduction

Program comprehension is a prerequisite to incremental change. A software developer who is tasked with changing a large software system spends effort on program comprehension activities to gain the knowledge needed to make the change [1]. For example, the developer spends effort to understand the system architecture or to locate the parts of the source code that implement the feature(s) being changed. Gaining such knowledge can be a time-consuming task, especially for developers who are unfamiliar with the system. Linguistic topics can help such developers to understand the system by revealing a latent structure that is not obvious from the package hierarchy or system documentation [2].

Linguistic topics are clusters of source code entities (e.g., classes) that are grouped by their natural language content (i.e., the words in their identifiers, comments, and literals). Such topics often correspond to the concepts and features implemented by the source code [3], and exploring such topics shows promise in helping developers to understand the entities that make up a system and to understand how those entities relate [2], [4]–[6]. Recent approaches to exploring linguistic topics in source code use machine learning (ML) techniques that model correlations among words, such as latent semantic indexing (LSI) [7] and latent Dirichlet allocation (LDA) [8], and ML techniques that also model correlations among documents, such as RTM [9].

Linguistic topics in source code have many applications in addition to general program comprehension. These applications include aspect mining [3] and traceability link recovery [10]. Yet, while researchers have used the extrinsic properties of topics in software engineering tasks, they have not yet measured their intrinsic properties. We believe that understanding these intrinsic properties will lead to a better understanding of how topics are implemented and thus will lead to a better understanding of how topics relate to each other and to source code entities such as packages or classes.

Ownership is one measurable intrinsic property of a topic. Ownership refers to whether a unit of development such as a source code entity, component, or linguistic topic is owned individually (i.e., by a single developer) or collectively (i.e., by multiple developers) and is measured by the number of changes made to the unit by each developer [11], [12]. Like exploring topics, examining ownership shows promise in helping developers to understand how a system's entities relate [12], [13]. For example, a developer can examine ownership to determine whom to contact with questions regarding the implementation of a topic.

In this paper we combine software repository mining and topic modeling to measure the ownership of topics in source code. We use the pachinko allocation model (PAM) [14] — specifically, four-level PAM — to extract linguistic topics from source code. Four-level PAM is a variant of LDA, and we use it because it models correlations among topics in addition to correlations among words. This allows us to compare intrinsic properties of similar topics. For example, in addition to asking questions about whether the classes in a linguistic topic share ownership characteristics, we can also ask questions about whether similar linguistic topics share ownership characteristics.

This paper makes the following contributions:

- An approach to modeling and measuring the ownership of linguistic topics in source code.
- A tool, `ohm`, that implements the approach for two programming languages (Java and C#).
- An exploratory study of the relationship between linguistic topics and ownership in source code.

We first review related work (§II). We next describe our methodology (§III) and our case study (§IV), which spans 10 open source Java systems. We then conclude (§V).

II. Background & Related Work

In this section we review background and related work.

A. Modeling Ownership in Source Code

We first distinguish between authorship and ownership. Authorship occurs when a developer writes a piece of code. By contrast, ownership occurs when a developer commits a new or changed piece of code to a source code repository. Authorship does not imply ownership and vice-versa.

The literature provides examples of expertise with regard to a piece of code being measured via development activity on that code. Fritz et al. [15] find that a developer's ability to answer questions about a piece of source code is determined by whether the developer has authored some of the code and by how much time the developer spent authoring the code. Further, Mockus and Herbsleb [16] present Expertise Browser, and McDonald and Ackerman [17] present a similar tool. Both tools recommend experts using measures of how much work different developers have put forth on pieces of code.

Pinzger et al. [18] use ownership networks to find fault prone binaries in Windows Vista. That is, they create networks in which a node representing a developer is connected to all binaries to which the developer has contributed and a node representing a binary is connected to all developers who have contributed to it. They apply social network metrics to these networks and find a strong relationship between the metrics and and post release failures. Bird et al. [19] later demonstrate that these metrics can predict failures in Eclipse as well.

Bird et al. [12] further adapt the Pinzger et al. [18] study, examining the effect of major and minor contributor edges. They define a major contributor as a developer who owns at least 5% of the changes to a software component. Similarly, they define a minor contributor as a developer who owns fewer than 5% of the changes to a software component. They find that information about minor contributors is key to defect prediction, because excluding minor contributor edges significantly decreases prediction power.

Rahman and Devanbu [13] use the provenance features of GIT to track the ownership of individual lines of source code. They then study the impacts of ownership and experience on software quality by comparing the ownership and experience characteristics of "implicated code" (lines of code changed to fix bugs) to those of "normal code." Rahman and Devanbu [13] report that strong ownership by a single developer is associated with implicated code and that lack of specialized experience on a particular file is associated with implicated code in that file. Similarly, in earlier work Mockus and Weiss [11] find that changes made by developers more experienced with a piece of code are less likely to cause a failure.

B. Pachinko Allocation Model

Latent Dirichlet allocation (LDA) [8] is a generative topic model. LDA models each document in a corpus of discrete data as a finite mixture over a set of topics and models each topic as an infinite mixture over a set of topic probabilities. That is, LDA models each document as a probability distribution indicating the likelihood that it expresses each topic and models each topic that it infers as a probability distribution indicating the likelihood of a word from the corpus being assigned to the topic. Yet, though LDA models correlations among words, it does not model correlations among topics.

The pachinko allocation model (PAM) [14], [20] is a family of generative topic models that build on LDA. PAM connects words and topics with a directed acyclic graph (DAG), where interior nodes represent topics and leaves represent words. PAM allows an arbitrary DAG to model topic correlations, but we use four-level PAM, in which the DAG is a four-level hierarchy with one root topic, with topics at the second level, with topics at the third level, and with words at the fourth level. The topics at the second level are called supertopics, and the topics at the third level are called subtopics. The DAG's root is connected to all supertopics, which are fully connected to subtopics, which are fully connected to words.

Four-level PAM uses a bag-of-words representation in which each document is a vector of counts with V components, where V is the size of the vocabulary. Inputs include the documents, the number of supertopics J, the number of subtopics K, and two Dirichlet hyperparameters α (for topic proportions) and β (for topic multinomials). Given these inputs, four-level PAM produces ϕ, the word-subtopic probability distribution, ψ, the subtopic-supertopic probability distribution, and θ, the subtopic-document probability distribution, and ι, the supertopic-document probability distribution. In particular, the distribution of the ith subtopic over V words is ϕ_i, the distribution of the jth supertopic over K subtopics is ψ_j, the distribution of the kth document over K subtopics is θ_k, and the distribution of the nth document over J supertopics is ι_n.

The results produced by PAM are immediately interpretable. We can examine ϕ to identify the most likely words in each subtopic (that is, the words with the highest probability of generating the subtopic) to determine the likely meaning of the subtopic. Similarly, we can examine ψ to identify the most likely subtopics in each supertopic — that is, the subtopics most likely to cooccur. We can examine θ to identify the most likely documents for each subtopic (that is, the documents with the highest probability of expressing the subtopic) to determine the likely members of the subtopic (i.e., the cluster). Similarly, we can examine ι to identify the most likely documents for each supertopic (that is, the documents with the highest probability of expressing the supertopic) to determine the likely members of the supertopic (i.e., the cluster of clusters).

In Table I we list an example four-level PAM-generated subtopic for Vuze[1], a bittorrent application. In particular, we list the 20 words with the highest probability of belonging to the subtopic. We can immediately interpret the results and determine that the subtopic is related to managing a network connection. In Table II we list the 10 classes with the highest probability of expressing the subtopic. Clearly, the classes are related to managing a network connection.

[1]http://vuze.com

TABLE I
TOP 20 WORDS IN A VUZE SUBTOPIC.

address	port	network	inet	tcp
socket	udp	proxy	admin	nat
host	protocol	bind	local	socks
addresses	server	interface	exception	http

TABLE II
TOP 10 CLASSES IN A VUZE SUBTOPIC.

core.networkmanager.admin.impl.NetworkAdminProtocolTester
core.networkmanager.admin.NetworkAdminASN
core.util.NetUtils
core.proxy.socks.impl.AESocksProxyAddressImpl
core.networkmanager.admin.NetworkAdminNATDevice
core.networkmanager.admin.NetworkAdminRoutesListener
core.networkmanager.admin.NetworkAdminProtocol
core.networkmanager.admin.NetworkAdminNetworkInterfaceAddress
net.natpmp.NatPMPDevice
core.networkmanager.admin.NetworkAdminNetworkInterface

C. Modeling Linguistic Topics in Source Code

Linstead et al. [21] use probabilistic Author-Topic (AT) modeling [22], an extension of LDA that captures the relationship of authors to topics in addition to the relationship of topics to documents, to extract the most likely authors for each topic. Their work is closely related to our own, as we consider the relationship between ownership and topics in source code. However, authorship and ownership are distinct concepts, as we describe in Section II-A. Further, Linstead et al. [21] use bug reports to attribute authorship to a single version of Eclipse (3.0), whereas we study ownership across the entire version history of multiple projects. Linstead et al. conclude that AT modeling produces "reasonable and interpretable automated topics and author-topic assignments," which they claim can "provide a basis for comparing the similarity of developers based on their contributions."

Kuhn et al. [4] present early work on exploring linguistic topics in source code, which they term semantic clustering. They use LSI and clustering to group packages, classes, and methods according to their linguistic similarities, where these clusters represent the topics in the source code. Maskeri et al. [5] identify topics in source code using LDA. However, the authors consider topics to be clusters of related terms used, rather than interpreting clusters as groups of linguistically similar source artifacts, as Kuhn did.

In more recent work Savage et al. [2] and Gethers et al. [6] use LDA and RTM, respectively, to explore linguistic topics in source code. Topic$_{\mathrm{XP}}$ [2] is a tool to visualize and search linguistic topics at the package, class, or method level of granularity. Similarly, CodeTopics [6] is a tool to visualize the similarity between source code and high-level artifacts, as well as the extent to which the source code covers topics extracted from those high-level artifacts.

III. METHODOLOGY

In this section we provide an overview of our methodology. We first describe the ownership model on which we base our work. We then describe the process by which we mine the data needed to build instances of this model. Next, we describe how we build instances of our linguistic model from source code and how we apply PAM to that model.

A. Ownership Model

Our methodology is based on the following ownership model. A *class* is a source code entity that has a core responsibility. Changes by developers can be traced to specific classes. A developer who commits (to a Subversion repository) a change to a class is a *contributor* to the class. A class c has a set of contributors $O(c) = \{o_1, \ldots, o_m\}$. A *system* is a set of classes $C = \{c_1, \ldots, c_n\}$, and a system C has a set of contributors:

$$O(C) = \bigcup_{c \in C} O(c)$$

The *ownership profile* for a class is a mathematical vector of length $|O(C)|$ that describes the frequency of changes to the class by each contributor to the system. That is, each cell in the vector corresponds to a contributor and contains the number of changes made to the class by that contributor. For example, consider class `Foo` in a system with five contributors. Assume that Developer 1 has changed `Foo` twice, Developer 2 has changed `Foo` once, Developer 4 has changed `Foo` sixteen times, and Developer 5 has changed `Foo` twelve times. The ownership profile for `Foo` is the five dimensional vector: $< 2, 1, 0, 16, 12 >$.

The ownership profile for a linguistic topic (i.e., for a four-level PAM subtopic) is a vector of length $|O(C)|$. It equals the vector sum of the ownership profiles for the classes that belong to the topic. That is, a linguistic topic is a cluster of classes, each of which has an ownership profile. We sum those profiles to model the ownership of the linguistic topic. For example, consider a topic containing three classes in a system with five contributors. Assume that the first class has ownership profile $< 4, 32, 9, 0, 0 >$, the second class has ownership profile $< 0, 2, 0, 0, 0 >$, and the third class has ownership profile $< 1, 0, 0, 2, 2 >$. The ownership profile for the topic is $< 5, 34, 9, 2, 2 >$.

Like Bird et al. [12] and Mockus & Weiss [11], we examine the number of changes to an entity made by a developer rather than the number of lines in an entity modified by a developer. As Bird et al. [12] observed, each change represents an exposure of the developer to the entity. Further, Elbaum and Munson [23] found a strong correlation between change frequency and change size, and Bird et al. [12] found similar results for Windows.

B. Source Code Repository Mining

Figure 1 illustrates the class diagram for ohm, our repository mining tool. ohm mines a source code repository and produces a database of ownership information. The Repository class wraps the access to a Subversion or CVS repository;

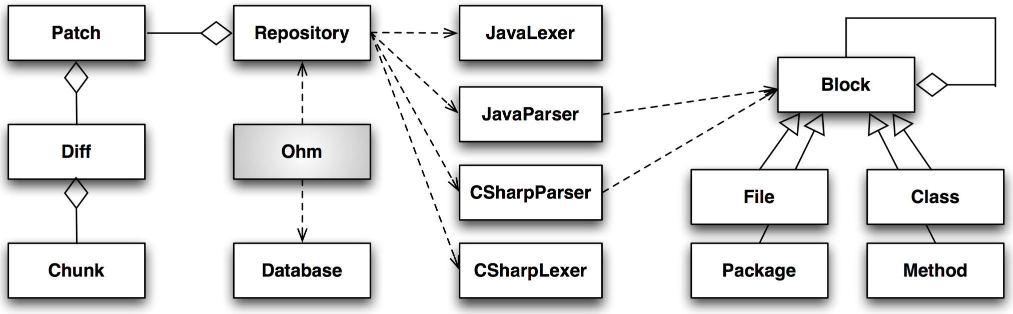

Fig. 1. Class diagram for ohm.

in the current implementation it uses *pysvn* to access a Subversion repository. Each Repository instance contains a list of Patch instances (i.e., commits) that together constitute the version history of the source code repository. Similarly, a Patch is a list of Diff instances, and a Diff is a list of Chunk instances. A Chunk represents a change and stores the file name and the start/end line numbers for the change. The Repository uses a JavaParser or CSharpParser to map each Chunk to one or more Block objects. In particular, Repository maps a Chunk to the innermost Block object that contains the line range, as well as to each Block object which (directly or indirectly) contains that innermost Block.

We build an ownership profile for each class in a software system by mining the complete version history of the system, from revision 1 to the HEAD revision. From each commit record in the Subversion repository for the subject system, we extract the name of the developer who made the commit and the names of the classes changed by the commit. We add this information to a database, and upon reaching the HEAD revision, we have a complete ownership profile for each class in the system. We adapt prior work [24] to map Subversion change logs to the names of classes. Our approach and tool are not specific to Subversion but are also compatible with CVS. Further, our tool handles both Java and C# input.

We track entity renames to improve the accuracy of our ownership profiles. Our algorithm for detecting renamed blocks is a modification of Dig et al. [25]. The four differences between our detection method and that of Dig are as follows:

1) Expanding on our previous work [24], we begin with two sets: R, which contains the found "removed" blocks from revision N, and A, which contains the found "added" blocks from revision $N + 1$. Note that this greatly reduces the number of pairwise comparisons.
2) We do not use the Shingles Encoding algorithm [26] to compare two blocks for similarity. We compare two blocks for similarity using a sequence matching technique provided by Python in the module *difflib*. The algorithm, originally developed by Ratcliff and Metzener [27], finds all of the longest common subsequences of the given two sequences. The comparison gives us a similarity ratio in the range [0, 1]. This ratio is defined as: ratio $= (2 * M)/T$, where M is the number of matches and T is the total number of elements in both sequences. Note that this is 1.0 if the sequences are identical, and 0.0 if they have nothing in common. For example, strings "abcd" and "bcde" have similarity ratio 0.75, or $(2 * 3)/8$.
3) We exploit our knowledge about the structure of the blocks. There are two ways in which we compare blocks in terms of sequences. First, if both b_R and b_A contain three or more sub-blocks (e.g., a class that has three methods), we use the sub-blocks as elements of each respective block's sequence. So, the sequence matcher will return a ratio based on the number of common sub-blocks. Second, if the first step finds that there are not enough sub-blocks to meet the requirements, we use the text of the blocks themselves. So, the sequence matcher will return a ratio based on the number of common lines (or strings).
4) After all comparisons between sets R and A are complete, we accept only the pairs which have the highest ratio. If there is a tie between two pairs, we will perform a tiebreaker by applying the same sequence matching technique to the names of the blocks to find the best possible ratio (i.e., the name which is most similar).

Like Dig et al. [25], we find that thresholds of the ratio between 0.5 and 0.7 produce the best results. Thus, we set the threshold of ratio $>= 0.6$ in our case study.

We also address the aliasing problem [28], though we currently do so manually. The aliasing problem occurs when there is not a one-to-one mapping between developers and Subversion accounts. In our case study, we noted and accounted for obvious aliases (e.g., `fredsa` and `fredsa@google.com`). Future improvements to ohm will include the integration of a state-of-the-art identity merge algorithm [29].

There are some potential practical problems with our approach to constructing ownership profiles. We give equal weight to all commits, but in practice there are many kinds of commits. For example, some commits include bug fixes, while others include only license changes. Ideally, the former kind of commit would be weighted differently than the latter. We also do not consider specific development practices such as branching and merging into the trunk [30].

C. Source Code Document Extraction

We use the following linguistic model. A *word* is the basic unit of discrete data in a software lexicon and is a sequence of letters. A *token* is a sequence of non-whitespace characters containing one or more words. An *entity* is a named source element such as a class, and an *identifier* is a token representing the name of an entity. *Comments* and *literals* are sequences of tokens delimited by language-specific markers (e.g., /* */ and quotes). The *document* which corresponds to a class is a sequence of words $d = (w_1, \ldots, w_m)$, and a *corpus* is a set of documents (i.e., classes) $D = (d_1, \ldots, d_n)$.

The left side of Figure 2 illustrates the source code document extraction process. A document extractor takes source code as input and produces a corpus as output. Each document in the corpus contains the words associated with a class. The text extractor is the first part of the document extractor. It parses the source code and produces a token stream for each class. With regard to Java, we consider an interface or enum name to be a class name. The preprocessor is the second part of the document extractor. It applies a series of transformations to each token and produces one or more words from the token. The transformations [31], [32]:

- *Splitting*: separate tokens into constituent words based on common coding style conventions (e.g., the use of camel case or underscores) and on the presence of non-letters (e.g., punctuation or digits)
- *Normalizing*: replace each upper case letter with the corresponding lower case letter
- *Filtering*: remove common words such as articles (e.g., 'an' or 'the'), programming language keywords, standard library entity names, or short words

We build a corpus from a single version of a software system. In particular, for the study we present in this paper, we use the HEAD revision of each subject system.

D. Topic Modeling

The right side of Fig 2 illustrates the topic modeling process. A topic modeling tool takes a corpus as input and produces linguistic topics (i.e., clusters) as output. We use Mallet[2] for topic modeling. Via analysis of the θ probability distribution we cluster the classes in the corpus. In particular, Mallet implements four-level PAM, which partitions documents (i.e., classes) into subtopics and supertopics. One of the outputs is the θ probability distribution, which for each document in the corpus lists the probability that the document belongs to each subtopic. We extract clusters by assigning each document to the subtopic to which it has the highest probability of belonging. Thus, each cluster corresponds to a subtopic and contains the classes most related to that subtopic. Not shown in the figure, we use the ψ probability distribution and a similar process to extract clusters of clusters. However, whereas we assign each document to exactly one subtopic, we (potentially) assign a subtopic to multiple supertopics. In particular, we assign the top 10 most likely subtopics to each supertopic.

[2]http://mallet.cs.umass.edu

IV. Case Study

In this section we describe the design of a case study in which we explore the relationship between ownership and linguistic topics in source code. We describe the case study using the Goal-Question-Metric approach [33].

A. Definition and Context

Our *goal* is to understand the relationship between ownership and linguistic topics in source code. The *quality focus* of the study is on informing development decisions and policy changes that could lead to software with fewer defects. The *perspective* of the study is of a researcher, developer, or project manager who wishes to gain understanding of the concepts or features implemented in the source code and to gain understanding of which developers are experts in these concepts or features. The *context* of the study spans the version histories of 10 open source Java systems.

Toward achievement of our goal, we pose the following research questions:

RQ1: Do classes that belong to the same linguistic topic have similar ownership characteristics?

RQ2: Do similar linguistic topics have similar ownership characteristics?

Basically, we want to know whether contributors own topics rather than just source code entities (i.e., classes). To answer these questions we use our ownership model.

In the remainder of this section we introduce the subjects of our study, describe the setting of our study, and report our data collection and analysis procedures.

1) Subject software systems: The 10 subjects of our study — Apache Ant[3], ArgoUML[4], CAROL[5], Google Web Toolkit[6] (GWT), iText[7], JabRef[8], jEdit[9], JHotDraw[10], Subversive[11], and Vuze— vary in size and application domain. Ant is a library and command-line tool for managing builds, ArgoUML is a UML modeling tool, CAROL is the common architecture for RMI ObjectWeb layer, GWT is a development toolkit for building browser-based applications, iText is a PDF manipulation library, JabRef is an open source bibliography reference manager, jEdit is a programmer's text editor, JHotDraw is a Java GUI framework, Subversive provides Subversion integration for Eclipse, and Vuze is a bittorrent client. Each system is stored in a Subversion repository, and the developers of each system use descriptive commit messages. Further, the developers store bug reports in an issue tracker.

We mined ownership data from the version history of the trunk of each system, from revision 1 to the HEAD revision. Table III lists the HEAD revision at which we stopped mining

[3]http://ant.apache.org
[4]http://argouml.tigris.org
[5]http://carol.objectweb.org
[6]http://code.google.com/webtoolkit
[7]http://itext.org
[8]http://jabref.sourceforge.net
[9]http://jedit.org
[10]http://jhotdraw.org
[11]http://subversive.org

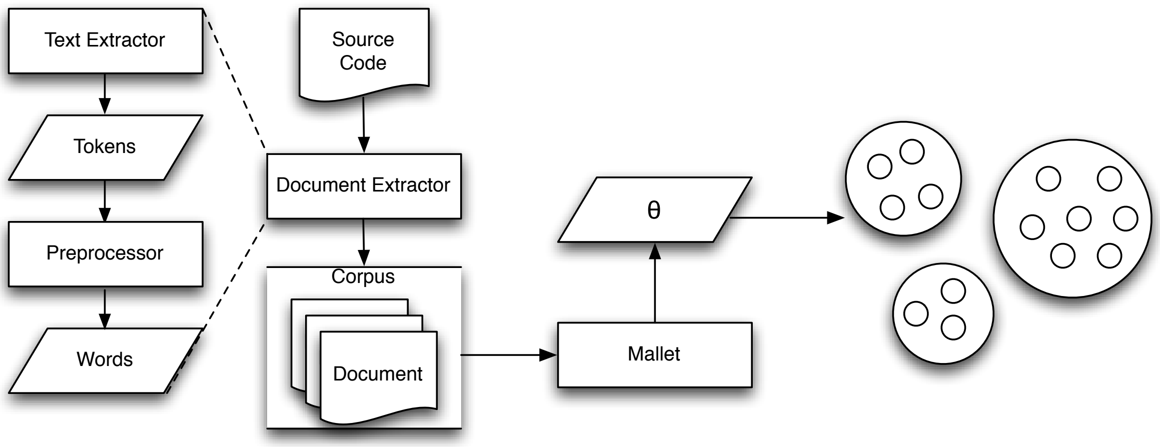

Fig. 2. The source code document extraction and topic modeling processes.

TABLE III
MINED REVISIONS.

	Ant	ArgoUML	CAROL	GWT	iText	JabRef	jEdit	JHotDraw	Subversive	Vuze
HEAD revision	1175200	19740	2214	10666	4979	3677	20000	768	21043	26757
Trunk revisions	12,589	17,627	1,222	5,626	4,443	2,712	4,600	657	1,237	22,667
Contributors	45	48	29	123	14	34	33	10	5	41

TABLE IV
SUBJECT SOFTWARE SYSTEMS.

	Ant	ArgoUML	CAROL	GWT	iText	JabRef	jEdit	JHotDraw	Subversive	Vuze
SLOC	127,860	176,059	13,652	648,912	79,845	117,431	110,696	139,856	104,194	502,683
CLOC	99,903	149,591	16,507	316,122	58,328	37,469	46,357	111,874	25,379	120,936
Classes	1,574	2,259	271	9,258	611	959	1,050	1,861	1,439	3,929
Words	7,489	6,910	1,747	25,906	6,545	6,076	4,979	5,260	2,989	10,917

ownership data, the total number of trunk revisions mined, and the total number of contributors. Table IV lists five size metrics for the HEAD revision of each subject system: source lines of code (SLOC), comment lines of code (CLOC), Java file count, class count, and (unique) word count. The class count includes named classes, interfaces, enums, and annotations.

2) Setting: To conduct the studies, we instantiate the process described in Section III.

We implemented our document extractor in Python v2.6 using ANTLR v3 and an open source Java 1.5 grammar[12]. We extract documents at the class level of granularity. We consider every class to be distinct. That is, if class `Bar` is nested within class `Foo`, each class is considered separately, and the text for class `Bar` is not considered to be part of the text for class `Foo`. We associate any comment that is contained in a class with that class. Further, like Fluri et al. [34], we associate any block comment (or series of line comments) that precedes a class with that class.

[12]http://antlr.org/grammar/1152141644268/Java.g

Our document preprocessor implements the three transformations described in Section III-C. We filter `java.lang` class names before splitting tokens. We split tokens based on camel case, underscores, and non-letters. We normalize to lower case before filtering English stop words [35], Java keywords, and words shorter than three characters.

We set J=25 and K=50 for CAROL, iText, jEdit, and JHotDraw, J=50 and K=100 for Ant, ArgoUML, JabRef, and Subversive, and J=100 and K=200 for GWT and Vuze. We empirically identified these values. Mallet v2.0 computes the four-level PAM models. The number of iterations is set to 1000, providing a balance between execution time and model convergence.

3) Data Collection and Analysis: We collected one ownership profile for each class in each of the 10 subject systems. We then used those ownership profiles to answer our two research questions. First, we used four-level PAM to partition the classes for each system into clusters (subtopics) and the clusters for each system into clusters of clusters (supertopics). To answer RQ1 we used cosine similarity to compare the ownership profiles for the classes in each subtopic. Specifically,

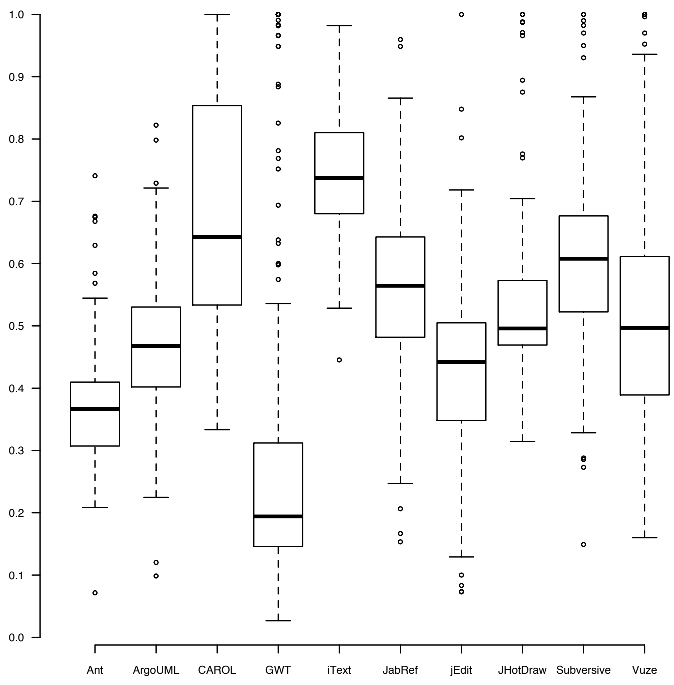

Fig. 3. Random cluster ownership similarity.

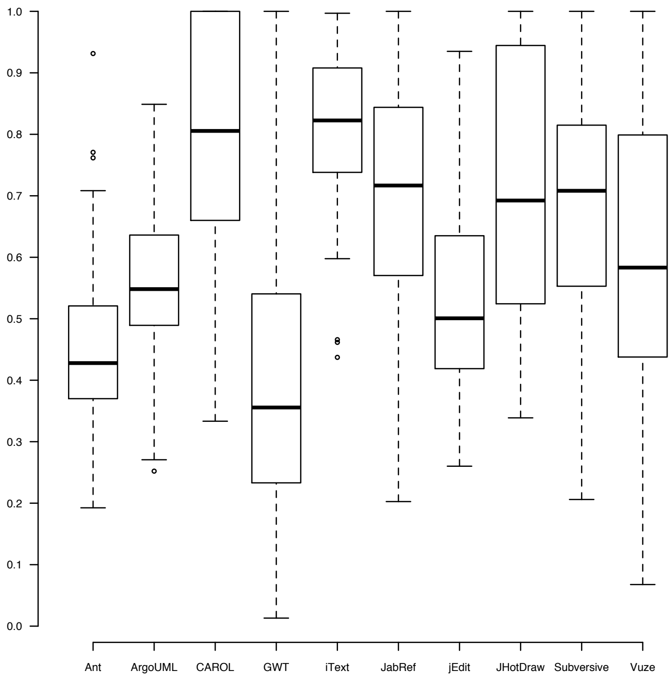

Fig. 4. Subtopic ownership similarity.

for each system we computed the average of the pairwise cosine similarity values for the classes in each subtopic. So, if we computed 50 subtopics for a system then we computed 50 average pairwise cosine similarity values for that system. To answer RQ2 we computed an ownership profile for each subtopic (see §III-A). We then used cosine similarity to compare the ownership profiles for the subtopics in each supertopic. Specifically, for each system we computed the average of the pairwise cosine similarity values for the subtopics in each supertopic. So, if we computed 25 supertopics for a system then we computed 25 average pairwise cosine similarity values for that system.

We also perform statistical analysis on our data. In particular, we use the Wilcoxon rank-sum test, a non-parametric statistical hypothesis test. We executed the tests using R[13].

B. Results

RQ1 asks whether classes that belong to the same linguistic topic (four-level PAM subtopic) have similar ownership characteristics. Before attempting to answer this question, we first established a baseline against which to compare. That is, to provide context for the answer to RQ1, we first generated random clusters of classes and measured the ownership characteristics of those random clusters. Thus, in addition to answering RQ1, we answer a closely related question: Are the ownership characteristics of classes that belong to the same linguistic topic more similar than the ownership characteristics of classes that belong to a randomly generated cluster? Figures 4 and 3 illustrate box plots that represent statistics describing the average of the pairwise cosine similarity values for the classes in each subtopic and random cluster, respectively. For example, the box plot for Ant represents 50 average pairwise cosine similarity values — one for the classes in each of the 50 subtopics (or random clusters) for Ant.

We used the Wilcoxon rank-sum test to determine whether the difference in similarity is significant. The test revealed a significant difference ($p < 0.001$) at the 95% significance level ($\alpha = 0.05$). Analogous tests for the 10 individual systems also indicate statistically significant differences. Thus, the answer to the question posed in the previous paragraph is yes, the ownership characteristics of classes that belong to the same linguistic topic are more similar than the ownership characteristics of classes that belong to a randomly generated cluster.

Again, RQ1 asks whether classes that belong to the same linguistic topic (four-level PAM subtopic) have similar ownership characteristics. The box plots in Figure 4 indicate that classes that belong to the same linguistic topic have similar ownership characteristics for many, but not all, of the systems. In particular, the median similarity values for 8 of the 10 subject systems are greater than 0.5. Though the median similarity value for GWT (the largest system and the one with the most contributors) is the lowest at less than 0.4, we observe no general relationship between the similarity and the size of the system or between the similarity and the number of contributors.

RQ2 asks whether similar linguistic topics have similar ownership characteristics. As with RQ1, before attempting to answer this question, we first established a baseline against which to compare. That is, to provide context for the answer

[13]http://r-project.org

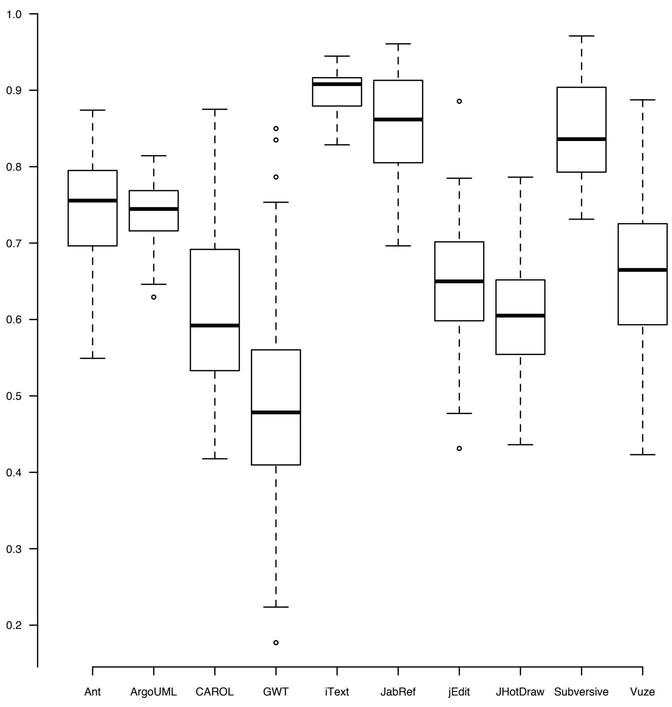

Fig. 5. Supertopic ownership similarity.

to RQ2, we first generated random clusters of subtopics and measured the ownership characteristics of those random clusters. Thus, in addition to answering RQ2, we answer a closely related question: Are the ownership characteristics of subtopics that belong to the same supertopic more similar than the ownership characteristics of subtopics that belong to a randomly generated cluster of subtopics?

We used the Wilcoxon rank-sum test to determine whether the difference in similarity is significant. The test revealed a significant difference ($p < 0.001$) at the 95% significance level ($\alpha = 0.05$). Analogous tests for the 10 individual systems also indicate statistically significant differences. Thus, the answer to the question is yes, the ownership characteristics of subtopics that belong to the same supertopic are more similar than the ownership characteristics of subtopics that belong to a randomly generated cluster of subtopics.

Again, RQ2 asks whether similar linguistic topics have similar ownership characteristics. The box plots in Figure 5 indicate that similar linguistic topics have similar ownership characteristics for most of the systems. In particular, the median similarity values for 9 of the 10 subject systems are greater than 0.5. As with RQ1, though the median similarity value for GWT is the lowest at just under 0.5, we observe no general relationship between the similarity and the size of the system or between the similarity and the number of contributors.

C. Threats to Validity

Our study has limitations that impact the validity of our findings, as well as our ability to generalize them. We describe some of these limitations and their impacts.

Threats to construct validity concern the adequacy of the study procedure with regard to measurement of the concepts of interest and can arise due to poor measurement design. Threats to construct validity include the use of average pairwise cosine similarity as our measure of similarity for topics/clusters. Other possible measures of similarity include the distance from the mean pairwise cosine similarity, but we think that average pairwise cosine similarity better captures the concept of interest.

Threats to internal validity include possible defects in our tool chain and possible errors in our execution of the study procedure, the presence of which might affect the accuracy of our results and the conclusions we draw from them. We controlled for these threats by testing our tool chain and by assessing the quality of our data. Because we applied the same tool chain to all subject systems, any errors are systematic and are unlikely to affect our results substantially.

An additional threat to internal validity relates to several subject systems having syntactically invalid Java files in their version histories. Because we could not parse these files, we could not mine ownership information for the classes contained within them. However, since less than 1% of the Java files that we encountered were syntactically invalid, these files are unlikely to affect our results substantially.

Another threat to internal validity pertains to the values of J and K that we selected for each subject system. These values affect the composition of the clusters whose ownership characteristics we measure. The literature provides no guidance regarding the selection of these values for four-level PAM, though Wei and Croft [36] suggest 50 to 300 topics as good general purpose values for LDA (which is equivalent to three-level PAM). Using these suggestions as a guide, we built multiple four-level PAM models for each subject system. We observed only negligible differences in our results, so we chose the final J and K values for each system based on the quality of the topics produced.

Threats to external validity concern the extent to which we can generalize our results. The subjects of our study comprise 10 open source Java systems, so we cannot generalize our results to systems implemented in other languages. However, the systems are of different sizes, are from different domains, and have characteristics in common with those of systems developed in industry.

D. Discussion

We posed two research questions with the goal of understanding the relationship between ownership and linguistic topics in source code. Specifically, we wanted to know whether contributors own topics rather than just source code entities such as classes. That is, we assume that it is typical for a contributor to view a software system as a collection of classes rather than as a collection of linguistic topics. Given such a view, a contributor would be likely to own certain classes. However, given that linguistic topics represent latent relationships among classes in a system, a contributor might also own topics (without explicit intention). Such ownership characteristics would lend further evidence to support the use of linguistic topics to understand a software system. In particular, if ownership of a system is already partitioned according to topics, we would have concrete evidence to support the notion that topics represent concepts and features implemented by the source code.

As noted by Baldi et al. [3], linguistic topics often correspond to the concepts and features implemented by the source code. In particular, Baldi et al. showed that topics correspond to crosscutting features (i.e., aspects). So, potential applications of our work include the easy identification of experts on crosscutting features or even the reallocation of development teams according to crosscutting features. Another potential application of our work is to improve bug triaging. That is, we could extract topics from a bug report and find the corresponding expert developer.

The results for RQ1 generally support the notion that classes that belong to the same linguistic topic have similar ownership characteristics. However, the results are stronger for certain systems than for others. Further, system characteristics such as size or number of contributors do not seem to explain the strength of the evidence. That is, consider the two systems with the largest similarity: CAROL and iText, both of which have similarity greater than 0.8. CAROL is an application, whereas iText is a library. CAROL has twice as many contributors as iText but also has half as many commits. In addition, CAROL is much smaller than iText in terms of SLOC, CLOC, Files, Classes, and Words.

Next, consider the two systems with the smallest similarity: Ant and GWT, both of which have similarity less than 0.5. Again, the two systems are from different domains, and GWT has nearly three times as many contributors as Ant. GWT is also much larger than Ant by all of our size measures. If we instead examine systems of similar size, we still do not see a trend emerge. For example, consider JabRef and jEdit, which are of similar size and which have similar numbers of contributors. The similarity for JabRef is greater than 0.7, whereas the similarity for jEdit is 0.5.

Though the results for RQ1 generally support an answer of "yes", more investigation is needed to understand why certain systems exhibit stronger evidence than others.

The results for RQ2 generally support the notion that similar linguistic topics have similar ownership characteristics. Indeed, the evidence for RQ2 is stronger than the evidence for RQ1. However, like for RQ1, the results for RQ2 are stronger for certain systems than for others. In addition, like for RQ1, system characteristics such as size or number of contributors do not seem to explain the strength of the evidence. Consider the three systems with the largest similarity: iText, JabRef, and Subversive, all of which have similarity greater than 0.8. iText is a library, JabRef is an application, and Subversive is an Eclipse plug-in. Though these three systems are those with the fewest contributors, we note that the two systems with the next largest similarity, Ant and ArgoUML, are those with the most contributors (other than GWT). Thus, we do not believe that number of contributors explains the amount of similarity.

Again, though the results for RQ2 generally support an answer of "yes", more investigation is needed to understand why certain systems exhibit stronger evidence than others.

V. Conclusion

In this paper we combined software repository mining and topic modeling to measure the ownership of linguistic topics in source code. We used the pachinko allocation model (PAM) — specifically, four-level PAM — to extract linguistic topics from source code. Four-level PAM is a variant of LDA, and we used it because it models correlations among topics in addition to correlations among words. This allowed us to compare properties of similar topics.

We addressed two research questions regarding the ownership of topics. First, we investigated whether classes that belong to the same linguistic topic have similar ownership characteristics. For 8 of our 10 subject systems we found evidence that suggests that classes belonging to the same topic do have similar ownership characteristics. However, we observed no general relationship between the size of the system and the amount of similarity or between the number of contributors and the amount of similarity. Next, we investigated whether similar linguistic topics have similar ownership characteristics. For 9 of our 10 subject systems we found evidence that suggests that similar topics do have similar ownership characteristics. Further, we observed that this evidence is stronger than the evidence for our first research question. However, we again observed no general relationship between the size of the system and the amount of similarity or between the number of contributors and the amount of similarity.

Future work includes expanding our study to include subject systems implemented in languages other than Java. It seems unlikely that our results are specific to Java systems, though we cannot confirm this assumption without experimentation. Additional future work includes an in-depth investigation of the development processes of each subject system to help understand why the topics in certain systems exhibit more similarity (with regard to ownership characteristics) than do others.

Acknowledgment

We thank the anonymous reviewers for their insightful comments and helpful suggestions. This material is based upon work supported by the U.S. Department of Education under Grant No. P200A100182 and by the National Science Foundation under Grant Nos. 0851824 & 0915559.

References

[1] T. Corbi, "Program understanding: challenge for the 1990's," *IBM Syst. J.*, vol. 28, no. 2, pp. 294–306, 1989.

[2] T. Savage, B. Dit, M. Gethers, and D. Poshyvanyk, "TopicXP: Exploring topics in source code using latent Dirichlet allocation," in *Proc. of 26th IEEE Int'l Conf. on Software Maintenance*, 2010.

[3] P. Baldi, E. Linstead, C. Lopes, and S. Bajracharya, "A theory of aspects as latent topics," in *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2008, pp. 543–562.

[4] A. Kuhn, S. Ducasse, and T. Girba, “Semantic clustering: Identifying topics in source code,” *Information and Software Technology*, vol. 49, no. 2007, pp. 230–243, 2007.

[5] G. Maskeri, S. Sarkar, and K. Heafield, “Mining business topics in source code using latent Dirichlet allocation,” in *Proc. of the 1st India Software Engineering Conference*, 2008, pp. 113–120.

[6] M. Gethers, T. Savage, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, “CodeTopics: Which topic am I coding now?” in *Proc. of the 33rd Int'l Conf. on Software Engineering*, 2011, pp. 1034–1036.

[7] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and H. R., “Indexing by latent semantic analysis,” *Journal of the American Society of Information Science*, pp. 391–407, 1990.

[8] D. Blei, A. Ng, and M. Jordan, “Latent Dirichlet allocation,” *J. of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[9] J. Chang and D. Blei, “Hierarchical relational models for document networks,” *Annals of Appl. Stats*, vol. 4, no. 1, pp. 124–150, 2010.

[10] H. Asuncion, A. Asuncion, and R. Taylor, “Software traceability with topic modeling,” in *Proc. of the 32nd Int'l Conf. on Software Engineering*, 2010, pp. 95–104.

[11] A. Mockus and D. Weiss, “Predicting risk of software changes,” *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[12] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, “Don't Touch My Code! Examining the Effects of Ownership on Software Quality,” in *Proc. of the ACM SIGSOFT Sym. on the Foundations of Software Engineering*, 2011.

[13] F. Rahman and P. Devanbu, “Ownership, experience and defects: a fine-grained study of authorship,” in *Proceeding of the 33rd Int'l Conf. on Software Engineering*, 2011, pp. 491–500.

[14] W. Li and A. McCallum, “Pachinko allocation: DAG-structured mixture models of topic correlations,” in *Proc. of the 23rd Int'l Conf. on Machine Learning*, 2006.

[15] T. Fritz, G. Murphy, and E. Hill, “Does a programmer's activity indicate knowledge of code?” in *Proc. of the ACM SIGSOFT Sym. on the Foundations of Software Engineering*, 2007, pp. 341–350.

[16] A. Mockus and J. Herbsleb, “Expertise browser: a quantitative approach to identifying expertise,” in *Proc. of the 24th Int'l Conf. on Software Engineering*, 2002, pp. 503–512.

[17] D. McDonald and M. Ackerman, “Expertise recommender: A flexible recomendation system and architecture,” in *Proc. of the ACM Conf. on Computer Supported Cooperative Work*, 2000.

[18] M. Pinzger, N. Nagappan, and B. Murphy, “Can developer-module networks predict failures?” in *Proc. of the ACM SIGSOFT Sym. on the Foundations of Software Engineering*, 2008, pp. 2–12.

[19] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, “Putting it all together: Using socio-technical networks to predict failures,” in *Proc. of the 17th Int'l Sym. on Software Reliability Engineering*, 2009.

[20] D. Mimno, W. Li, and A. McCallum, “Mixtures of hierarchical topics with pachinko allocation,” in *Proc. of the 24th Int'l Conf. on Machine Learning*, 2007.

[21] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, “Mining eclipse developer contributions via author-topic models,” in *Proc. of the Int'l Conf. on Software Engineering Workshops*, 2007.

[22] M. Steyvers, P. Smyth, M. Rosen-Zvi, , and T. Griffiths, “Probabilistic author-topic models for information discovery,” in *Proc. of the 10th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, 2004, pp. 306–315.

[23] S. Elbaum and J. Munson, “Code churn: A measure for estimating the impact of code change,” in *Proc. of the Int'l Conf. on Software Maintenance*, 1998.

[24] C. Corley, N. Kraft, L. Etzkorn, and S. Lukins, “Recovering traceability links between source code and fixed bugs via patch analysis,” in *Proc. of the 6th Int'l Wksp. on Traceability in Emerging Forms of Software Engineering*, 2011, pp. 31–37.

[25] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, “Automated detection of refactorings in evolving components,” in *ECOOP 2006 Object-Oriented Programming*, ser. Lecture Notes in Computer Science, D. Thomas, Ed. Springer Berlin / Heidelberg, 2006, vol. 4067, pp. 404–428.

[26] A. Broder, “On resemblance and containment of documents,” in *Proc. of SEQUENCES*, 1997.

[27] J. Ratcliff and D. Metzener, “Pattern matching: The gestalt approach,” *Dr. Dobb's Journal*, 1988. [Online]. Available: http://drdobbs.com/database/184407970?pgno=5

[28] G. Robles and J. Gonzalez-Barahona, “Developer identification methods for integrated data from various sources,” in *Proc. of the Int'l Wksp. on Mining Software Repositories*, 2005.

[29] M. Göminne and T. Mens, “A comparison of identity merge algorithms for software repositories,” *Science of Computer Programming*, 2011.

[30] C. Williams and J. Spacco, “Branching and merging in the repository,” in *Proc. of the Int'l Wksp. on Mining Software Repositories*, 2008.

[31] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, “An information retrieval approach to concept location in source code,” in *Proc. of the 11th Working Conf. on Reverse Engineering*, 2004, pp. 214–223.

[32] A. Marcus and T. Menzies, “Software is data too,” in *Proc. of the FSE/SDP Wksp. on Future of Software Engineering Research*, 2010, pp. 229–232.

[33] V. Basili, G. Caldiera, and H. Rombach, “The goal question metric approach,” 1994. [Online]. Available: ftp://ftp.cs.umd.edu/pub/sel/papers/gqm.pdf

[34] B. Fluri, M. Wursch, and H. Gall, “Do code and comments co-evolve? On the relation between source code and comment changes,” in *Proceedings of the 14th Working Conference on Reverse Engineering*, 2007, pp. 70–79.

[35] C. Fox, “Lexical analysis and stoplists,” in *Information Retrieval: Data Structures and Algorithms*, W. Frakes and R. Baeza-Yates, Eds. Prentice-Hall, 1992.

[36] X. Wei and W. Croft, “LDA-based document models for ad-hoc retrieval,” in *Proc. of ACM SIGIR*, 2006.